University of Alberta

TRANSPARENT DATAFLOW DETECTION AND USE IN WORKFLOW SCHEDULING: CONCURRENCY AND DEADLOCK AVOIDANCE

by

Yang Wang

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Fall 2008

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-46450-2
Our file Notre référence
ISBN: 978-0-494-46450-2

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

To my wife, Liang Lei.

Abstract

In this thesis we demonstrate the value of dataflow information to improve makespan performance (i.e., time to complete a set of jobs) in batch-scheduled workloads. Novel mechanisms and policies are introduced to improve job concurrency (i.e., when resources are unlimited) and to reduce the impact of deadlock (i.e., when resources are constrained). Without dataflow information concurrency might be limited, even if resources are unlimited, and resource usage might be inefficient, even if resource utilization is superficially high. The key insight is that dataflow, unlike control-flow, makes it visible when resources can be deallocated or reallocated, which allows for a crucial distinction between *active* and *inactive* resource usage. Through a simulation study, we show that the benefits of dataflow information can be a reduced makespan of over a factor of 5, depending on the workload and available resources.

Despite a large body of research on dataflow, most high-performance computing (HPC) systems (e.g., clusters) are batch scheduled based on control-flow. The lack of a simple way to obtain dataflow information and the lack of compelling policies to exploit dataflow may account for the control-flow status quo. Therefore, we describe a simple prototype for transparently gathering dataflow information (i.e., *Workflow-aware File System* (WaFS)) and several scheduling policies to exploit that knowledge for higher concurrency (e.g., *Versioned Namespace* (VNS), *Overwrite-Safe Concurrency* (OSC)) and for better deadlock handling (e.g., *Dataflow Aggregate Requests* (DAR), *Dataflow Topological Ordering* (DTO)). Notably, our simulations show how dataflow information allows our policies to have lower makespans than the classic banker's algorithm and Lang's algorithm.

Acknowledgements

It has been a longer journey than I originally expected to finish the dissertation. However, it has not been a solitary journey. Many people have provided supports to me along the way. I owe a great many of thanks to them.

First, I would like to thank my supervisor, Dr. Paul Lu, for his guidance, his patience and his understanding during this challenging time. Without his help and support, my dissertation would not have been possible. I would also like to thank other members of my supervisory committee, Dr. Michael MacGregor, Dr. Jonathan Schaeffer, Dr. Vincent Gaudet and Dr. Mario Nascimento, for their time and energy in reviewing my proposal and their invaluable advice and feedback on my research.

Thanks also go to Cam Macdonell, Zhuang Guo, Meng Ding, Mike Closson, Nicholas Lamb, Paul Nalos, and all the other members of the Trellis group for their excellent work in Trellis and for all the help they have given me.

In addition, I am extremely grateful for the scholarship and financial supports from the National Sciences and Engineering Research Council (NSERC) of Canada, the Informatics Circle of Research Excellence (iCORE), and the University of Alberta who provided the funding to allow me to finish this dissertation.

Last, but not least, this dissertation would not be possible without the moral support and encouragement of my family who have been with me throughout this long journey.

Contents

1 Introduction 1
- 1.1 Research Goal . . . 1
  - 1.1.1 The Challenges . . . 2
  - 1.1.2 The Advantages of Dataflow Information . . . 5
- 1.2 Contributions . . . 6
- 1.3 Dissertation Organization . . . 8

2 Background Knowledge 10
- 2.1 Basic Concepts . . . 10
- 2.2 Deadlock and the Banker's Algorithm . . . 13
- 2.3 Performance Metrics . . . 14
- 2.4 Scheduling in Batch Queuing Systems . . . 16
- 2.5 Some Typical Workflow Applications . . . 19
  - 2.5.1 GROMACS: Molecular Dynamics . . . 19
  - 2.5.2 Proteome Analyst (PA): Bioinformatics, Machine Learning . . . 20
  - 2.5.3 Bronze Standard Medical Imaging (BSMI): Medical Image Processing . . . 21
- 2.6 Concluding Remarks . . . 22

3 Dataflow Collection and Scheduling Policies 24
- 3.1 Motivation . . . 24
- 3.2 Dataflow Collection: WaFS Scheduler . . . 26
- 3.3 Dataflow-based Scheduling Policies . . . 27
  - 3.3.1 Versioned Namespace (VNS) Policy . . . 29
  - 3.3.2 Overwrite-Safe Concurrency (OSC) Policy . . . 29
  - 3.3.3 Hybrid Policy (HB) . . . 30
  - 3.3.4 Summary . . . 31
- 3.4 Simulation Results . . . 32
  - 3.4.1 Methodology . . . 32
  - 3.4.2 Results, Data Points and Standard Deviation . . . 34
  - 3.4.3 Summary . . . 42
- 3.5 Concluding Remarks . . . 45

4 Dataflow-based Scheduling for Deadlock Avoidance 46
- 4.1 Notation and Workflow Model . . . 47
- 4.2 The Algorithms . . . 49
  - 4.2.1 The DAR Algorithm . . . 50
  - 4.2.2 The DTO Algorithm . . . 54
  - 4.2.3 Summary . . . 57
- 4.3 Deadlock Avoidance Batch Scheduler . . . 58
  - 4.3.1 The Priority-based Batch Scheduler . . . 59
  - 4.3.2 Integration with Deadlock Avoider . . . 60
- 4.4 Active-Instance-Aware Admission Control . . . 62
  - 4.4.1 The Admission Control Algorithm . . . 64

4.5 Simulation Results . . . 66
4.5.1 Methodology . . . 67
4.5.1.1 Benchmark Workloads . . . 67
4.5.1.2 Reference Algorithms . . . 67
4.5.1.3 Simulated Platforms . . . 68
4.5.1.4 Instance Admission Control . . . 68
4.5.2 Performance Metrics . . . 69
4.5.3 Organization . . . 69
4.5.4 Data Points and Standard Deviation . . . 70
4.5.5 Sensitivity to Workload Characteristics . . . 70
4.5.5.1 Sensitivity to Workflow Shapes: Performance Changes Depend on the Shapes . . . 71
4.5.5.2 Insensitivity to Workflow Shape Parameters: Performance Changes Are Not Sensitive . . . 81
4.5.5.3 High Sensitivity to Workflow Sizes: Performance Differences Are Enlarged . . . 83
4.5.5.4 Insensitivity to Job Characteristics: Performance Remains Largely Unchanged . . . 91
4.5.5.5 Sensitivity to File Access Patterns: Relative Performance is Altered 92
4.5.5.6 Summary . . . 96
4.5.6 Comparison with Deadlock Detection . . . 96
4.5.7 Performance Benefits of Instance Admission Control . . . 104
4.6 Concluding Remarks . . . 109

5 WaFS Prototype **113**
5.1 Design Options . . . 113
5.2 The Batch Scheduler and Versioned Namespace Manager . . . 114
5.3 The Monitor . . . 115
5.3.1 Handling open() and close() . . . 116
5.3.2 Manipulating the File Pathname . . . 116
5.3.3 Tracing Process Family . . . 117
5.4 Proof-of-Concept and Results . . . 118
5.4.1 Dataflow Collection: A Running Example . . . 118
5.4.2 Overhead of Prototype on Potential Applications . . . 120
5.4.3 Simulation Results for an GROMACS Workload . . . 122
5.5 Concluding Remarks . . . 125

6 Related Work **126**
6.1 Filename Conflict Resolution . . . 126
6.2 Related File System Studies . . . 127
6.3 Dataflow Applications . . . 128
6.4 Storage-Aware Workflow Scheduling . . . 130
6.5 Deadlock Avoidance . . . 132

7 Concluding Remarks **135**
7.1 Limitations . . . 135
7.2 Conclusions . . . 136
7.3 Future Work . . . 137

Bibliography **138**

A Comparisons with Lang's algorithm in Workflow-Based Computation **145**
A.1 An Overview of Lang's Algorithm . . . 145
A.2 The Problems of Lang's Algorithm in Workflow-based Computation . . . 147
A.3 The Comparisons between Lang's Algorithm and our Algorithms: DAR and DTO . 149

B Non-Uniform Distributions: Zipf-based Workloads **150**
B.1 Average of Simulated Makespans 150
B.2 Median of Simulated Makespans 151

List of Figures

1.1 Proteome Analyst Workflow (a 6-job Fork&Join workflow) 2
1.2 Control-flow is not always safe to use in exploiting inter-workflow instance concurrency. An example (a) shows how File Out.A is unsafely overwritten by Job A in WI2 before Job D in WI1 consumes it (The data dependency between Job A and Job D is not shown in the control-flow). The dataflow example (b) is correct in that both Job C and Job D in WI1 must complete before Job A in WI2 can overwrite File Out.A. Note that the dataflow from Job A to Job D is only specific to (b) for comparison purposes. 3

2.1 An Example of a Fork&Join Workflow 11
2.2 An Example Input Script File for DAGMan 11
2.3 The Banker's Algorithm for Requesting a Resource Allocation 14
2.4 The Safety Checking Algorithm in the Banker's: The algorithm is performed to find out if the system is in a safe state. 15
2.5 An Architecture of a Typical Batch Queuing System: A central job queue is maintained by Queue Manager, which performs some particular scheduling algorithm to map jobs to a set of interconnected computers. 17
2.6 An Example of Typical GROMACS Workflow Chart 19
2.7 An Example of a Typical PA Workflow Chart 21
2.8 An Example of a Bronze Standard Medical Imaging Workflow Chart (a). We view it as a Lattice-like workflow shape (b) in our discussion. 22

3.1 WaFS Scheduler: Integration of WaFS with Batch Scheduler for Dataflow Collection 26
3.2 Inter-Workflow Instance Concurrency: (a) Serial Policy (BASE), (b) Versioned Namespace (VNS) and (c) Overwrite-Safe Concurrency (OSC) 28
3.3 Inter-Workflow Instance Concurrency in HB Policy 31
3.4 Benchmark Workflow Graphs: A circle represent a job, and a rounded rectangle represents an input/output file. The Fork&Join (a) is characterized by the fan-out factor and the number of stages, whereas the Lattice (b) is characterized by its height and width. 32
3.5 Simulation Results for the Fork&Join (3×32): (a) Makespan, (b) Average DOC and (c) Storage Overhead. (DOC units are numbers of jobs; all other values are either time units or storage units) 35
3.6 Simulation Results for the Lattice (8×12): (a) Makespan, (b) Average DOC and (c) Storage Overhead. (DOC units are numbers of jobs; all other values are either time units or storage units) 37
3.7 Simulation Results for the Pipeline (10-stage): (a) Makespan, (b) Average DOC and (c) Storage Overhead. (DOC units are numbers of jobs; all other values are either time units or storage units) 39
3.8 Impacts of Job Service Time on the Fork&Join (3×12): Makespan, Average DOC and Storage Overhead (Left: JST[10, 1000], Right: JST[800, 1000]). 41
3.9 Impacts of Job Service Time on the Lattice (8×12): Makespan, Average DOC and Storage Overhead (Left: JST[10, 1000], Right: JST[800, 1000]). 43

3.10 Impacts of Job Service Time on the Pipeline (10-stage): Makespan, Average DOC and Storage Overhead (Left: JST[10, 1000], Right: JST[800, 1000]). 44

4.1 An example showing how the maximum claims defined in DAR are computed. In this example three workflow instances I_is with different shapes, (a) Pipeline, (b) Fork&Join and (c) Lattice, are considered. The nodes in the graphs represent the jobs, and the jobs inside the dashed regions are those jobs that have not been completed as of time t'. The numbers marked beside each edge indicate the file sizes. . 51
4.2 The DAR Algorithm. 53
4.3 The DTO Algorithm 56
4.4 The Safety Checking Algorithm in DTO: The algorithm is performed to find out if a job scheduling in instance I_i is in a safe state. 57
4.5 The topological_order algorithm in the DTO algorithm 58
4.6 Deadlock Avoidance Batch Scheduler. MDF stands for Most Done Job First, an Instance Scheduling Policy. 59
4.7 Algorithm for Dealing with New Instance Arrivals 61
4.8 Algorithm for Dealing with Job Completions 62
4.9 Algorithm for Triggering Batch Scheduling 63
4.10 The Deadlock Avoider Algorithm 63
4.11 An example illustrating that allocating storage without admission control may incur poor performance. 64
4.12 The Extended Deadlock Avoider after Integrating with Instance Admission Control 66
4.13 Classification of Storage Resource Utilization 69
4.14 Impacts of Workflow Shape on the Makespans of the Compared Algorithms when the Workflow Sizes are Small 72
4.15 How the storage is used by each of compared algorithms for Fork&Join (3×8). . . 74
4.16 How the storage is used by each of the compared algorithms for Lattice (4×6). . . 75
4.17 How the storage is used by each of the compared algorithms for Pipeline (5-stage). 77
4.18 Execution Traces (DAR): The total number of workflow instances that are admitted and completed by DAR as the computation proceeds, given the storage budget of 250 units for the Fork&Join (3×8), 1200 units for the Lattice (4×6), and 50 units for the Pipeline (5-stage). 78
4.19 Execution Traces (DTO): The total number of workflow instances that are admitted and completed by DTO as the computation proceeds, given the storage budget of 250 units for the Fork&Join (3×8), 1200 units for the Lattice (4×6), and 50 units for the Pipeline (5-stage). 79
4.20 Impacts of Workflow Shape Parameters on the Makespans of the Compared Algorithms for the Fork&Join (26 jobs): Note that Figure 4.20(c) is identical to Figure 4.14(a). 81
4.21 Impacts of Workflow Shape Parameters on the Makespans of the Compared Algorithms for Lattice (24 jobs): Note that Figure 4.21(c) is identical to Figure 4.14(b) with different scales along the y-axis. 82
4.22 Impacts of Workflow Size on the Makespans of the Compared Algorithms when the Workflow Sizes become Large 84
4.23 Execution Traces (Lattice (8×12)): The total number of workflow instances that are admitted and completed by DAR and DTO as the computation proceeds, given a storage budget of 1100 units for the Lattice (8×12). 85
4.24 Execution Traces (Lattice (4×6)): The total number of workflow instances that are admitted and completed by DAR and DTO as the computation proceeds, given a storage budget of 300 units for the Lattice (4×6). 86
4.25 How the storage is used by each of the compared algorithms for the Fork&Join (3×32). 88
4.26 How the storage is used by each of the compared algorithms for the Lattice (8×12). 89

4.27 How the storage is used by each of the compared algorithms for the Pipeline (10-stage). 90
4.28 Impacts of File Size Distribution Parameters on the Makespans of the Compared Algorithms: Figure 4.28(a) is the same as Figure 4.22(c). 91
4.29 Makespan Comparisons: The algorithms are compared for the Fork&Join (3×32) workload with a file access pattern such that a single output file of the first job is read by all its child jobs (i.e., Fork&Join$^+$ (3×32)). 93
4.30 Concurrency Comparisons: The algorithms are compared for the Fork&Join$^+$ (3×32) with a multiple-reader access pattern. 94
4.31 Concurrency Comparisons: The algorithms are compared for the Fork&Join (3×32) with a single-reader access pattern. 95
4.32 Execution Traces: The total number of workflow instances are admitted and completed by the compared algorithms as the computations proceed. The examined workflow is the Fork&Join$^+$ (3×32) with a multiple-reader access pattern, and the storage budget is 1400 units. 97
4.33 Makespan Comparisons (Small Workflow Size): The deadlock avoidance algorithms and detection algorithm Det(0.5Bgt_LDF) are compared when the workflow shape is changed. Each file has only one reader. 98
4.34 Makespan Comparisons (Large Workflow Size): The deadlock avoidance algorithms and detection algorithm Det(0.5Bgt_LDF) are compared when the workflow shape is changed. Each file in the workloads has only one reader. 99
4.35 Makespan Comparisons: the deadlock avoidance algorithms and detection algorithm Det(0.5Bgt_LDF) are compared for the Fork&Join$^+$ (3×8) and Fork&Join$^+$ (3×32) with a multiple-reader access pattern. 102
4.36 Performance Benefits of Instance Admission Control (IAC) Measured by the Normalized Makespan (Small Workflow Size) 105
4.37 Performance Benefits of Instance Admission Control (IAC) Measured by the Normalized Makespan (Large Workflow Size) 106
4.38 Performance Benefits of Instance Admission Control (IAC) Measured by the Normalized Makespan: DAR, DTO and the banker's algorithm are compared in the absence and presence of IAC for the benchmark workloads of Fork&Join$^+$ (3×8) and Fork&Join$^+$ (3×32) with a multiple-reader access pattern. 112

5.1 An Example of a GROMACS Input Script: gromacs.st 120
5.2 The Given CFG 121
5.3 The Detected DFG 121
5.4 Performance of GROMACS *gmxbench* in absence and presence of WaFS. The performance overhead of WaFS is shown by the percentage above the bar. 122
5.5 Simulation Results for the GROMACS Workload: Compared to Figure 4.33(c) and 4.34(c), DTO shows greater performance advantages compared to other policies. . . 125

6.1 An Example of Condor Submit Description File 127

A.1 Resource-request Graphs used in Lang's Algorithm: To compare with the storage utilization in workflow-based computation, only a single type of resources is depicted. In fact, Lang's algorithm can handle multiple types of resources. 146
A.2 An Example of Workflow DAG and its Scheduling Graph 148

B.1 Histogram of 1,000 Zipf Distributed Numbers ($\alpha = 1.5$, maximum=500) 151
B.2 Average Makespans: Zipf Distributions ($\alpha = 1.5$) for Job Service Time and File Size (10 Runs, Average) 152
B.3 Median Makespans: Zipf Distributions ($\alpha = 1.5$) for Job Service Time and File Size (10 Runs, Median) 153

List of Tables

1.1 The Characteristics of the Proposed Policies and Associated Deadlock Avoidance Algorithms: BASE/Serial and Sub-dir correspond to our discussed serial policy and the policy that employs the sub-directory-based strategy to address the filename conflict problem. VNS, OSC and HB are our proposed policies. The HB policy contains several deadlock avoidance algorithms among which DAR and DTO are our proposed deadlock avoidance algorithms. Banker's and Lang's are reference algorithms for evaluating both DAR and DTO in the later chapters. 9

2.1 Some Global Variables Used in the Banker's Algorithm: n is the number of processes that are involved in deadlock avoidance. 13
2.2 Three Typical Workflow Applications: GROMACS, Proteome Analyst (PA) and Bronze Standard Medical Imaging (BSMI) 20

3.1 The Characteristics of the Compared Policies: VNS, OSC and HB are our dataflow-based policies, BASE is the control-flow-based serial policy, and Sub-dir refers to the policy that employs the working directory to address the filename conflicts and maximize the job concurrency. BASE and Sub-dir policies are listed for comparison purposes. DOC is short for "Degree of Concurrency." 27
3.2 The Characteristics of the Benchmark Workloads 34

4.1 Notation Used in Algorithm Descriptions 48
4.2 The Characteristics of the Compared Deadlock Avoidance Algorithms 49
4.3 The computation of the maximum claims are detailed for the three examined workflow instances at $t = 0, t'$ where, as of time t', some jobs have been completed in each instance. 52
4.4 Scheduling States, Events and Corresponding Actions 60
4.5 Notation Used in Instance Admission Control 65
4.6 The Distribution of the Standard Deviations of the Makespan Data in Our Simulations 70
4.7 Investigated Workflow Shape Parameters: The total number of jobs is fixed as 24 for Lattice and 26 for Fork&Join (two extra nodes for the source and sink). * indicates the shape parameters that were studied in the previous experiments. 80
4.8 Comparisons of the Overhead of Deadlock Recovery between Lattice (4×6) and Lattice (8×12): Note that the storage budget given to the Lattice (8×12) is 4 times as much as the budget given to the Lattice (4×6). 100
4.9 Comparisons of the Overhead of Deadlock Recovery between Pipeline (5-stage) and Pipeline (10-stage): Note that the storage budget given to the Pipeline (10-stage) is twice as much as the budget given to the Pipeline (5-stage). 101
4.10 Performance Benefits of Instance Admission Control (IAC) Measured by Makespan: The algorithms are compared (measured in time units) for the Fork&Join (3×8) workload when IAC is present and when it is absent. Storage budget is varied from 250 to 1450 storage units. The lowest makespan in each row is boldfaced. 108

4.11 Performance Benefits of Instance Admission Control (IAC) Measured by Makespan: The algorithms are compared (measured in time units) for the Lattice (4×6) workload when IAC is present and when it is absent. Storage budget is varied from 400 to 2000 storage units. The lowest makespan in each row is boldfaced 109
4.12 Performance Benefits of Instance Admission Control (IAC) Measured by Makespan: The algorithms are compared (measured in time units) for the Pipeline (5-stage) workload when IAC is present and when it is absent. Storage budget is varied from 50 to 300 storage units. The lowest makespan in each row is boldfaced. 109
4.13 Performance Benefits of Instance Admission Control (IAC) Measured by Makespan: The algorithms are compared (measured in time units) for the Fork&Join$^+$ (3×8) workload (multiple readers) when IAC is present and when it is absent. Storage budget is varied from 250 to 1450 storage units. The lowest makespan in each row is boldfaced. 110
4.14 Performance Benefits of Instance Admission Control (IAC) Measured by Makespan: The algorithms are compared (measured in time units) for the Fork&Join (3×32) workload when IAC is present and when it is absent. Storage budget is varied from 1000 to 2400 storage units. The lowest makespan in each row is boldfaced. 110
4.15 Performance Benefits of Instance Admission Control (IAC) Measured by Makespan: The algorithms are compared (measured in time units) for the Lattice (8×12) workload when IAC is present and when it is absent. Storage budget is varied from 1200 to 3400 storage units. The lowest makespan in each row is boldfaced. 111
4.16 Performance Benefits of Instance Admission Control (IAC) Measured by Makespan: The algorithms are compared (measured in time units) for the Pipeline (10-stage) workload when IAC is present and when it is absent. Storage budget is varied from 100 to 600 storage units. The lowest makespan in each row is boldfaced. 111
4.17 Makespan Comparison: The algorithms are compared (measured in time units) for the Fork&Join$^+$ (3×32) workload (multiple readers) when IAC is present and when it is absent. Storage budget is varied from 1000 to 2400 storage units. The lowest makespan in each row is boldfaced. 111

5.1 Experimental Configuration for *gmxbench* 122
5.2 WaFS-measured Job Service Times and File Sizes for the GROMACS *S-peptide* Workflow 124
5.3 WaFS-measured Job Service Times and File Sizes for the `6.mdrun` stage of GROMACS *gmxbench* 124

6.1 Comparison between Bent's System, Ramakrishnan's System and the WaFS Scheduler 131
6.2 Comparison of Some Banker's-based Deadlock Avoidance Algorithms (m: the number of resource types, n: the number of processes, r: the number of resource units) . 132

A.1 Comparisons between Lang's Algorithm and our Algorithms: DAR and DTO . . . 149

B.1 Standard Deviations for Figure B.2(a) 154
B.2 Standard Deviations for Figure B.2(b) 155
B.3 Standard Deviations for Figure B.2(c) 156

List of Abbreviations

Abbreviation	Full Name	Definition and Discussion
BSMI	Bronze Standard Medical Imaging	Chapter 2.5
CFG	Control-flow Graph	Chapter 2.1
CP	Critical Path	Chapter 4.3.1
DAG	Directed Acyclic Graph	Chapter 2.1
DAR	Dataflow-based Aggregate Requests	Chapter 1.2, Chapter 4.2.1
DFG	Dataflow Graph	Chapter 2.1
DOC	Degree of Concurrency	Chapter 2.3
DTO	Dataflow-based Topological Ordering	Chapter 1.2, Chapter 4.2.2
FCFS	First Come First Serve	Chapter 2.4
FMS	Flexible Manufacturing System	Chapter 6.5
GEL	Grid Execution Language	Chapter 6.1
GFS	Google File System	Chapter 6.2
HB	Hybrid	Chapter 3.3.3, Chapter 4
HLF	Highest Level First	Chapter 4.3.1
HPC	High Performance Computing	Chapter 1.1
IAC	Instance Admission Control	Chapter 4.4.1, Chapter 4.5.7
ICP	Iterative Closet Point	Chapter 2.5
JCML	Job Control Markup Language	Chapter 4.3.1
JST	Job Service Time	Chapter 3.4
LDF	Least-Done Job First	Chapter 4.5.6
LFS	Log Structure File System	Chapter 6.2
LKM	Loadable Kernel Module	Chapter 5
LPT	Longest Processing Time	Chapter 4.3.1
LP	Longest Path	Chapter 4.3.1
LiFS	Linking File System	Chapter 6.2

MDF	Most-Done Job First	Chapter 4.3.1
MD	Molecular Dynamics	Chapter 2.5, Chapter 5.4.1
MPSoC	MultiProcessor System-on-a-Chip	Chapter 6.5
MRT	Mean Response Time	Chapter 2.3
MS	Makespan	Chapter 2.3
OSC	Overwrite-Safe Concurrency	Chapter 1.2, Chapter 3.3.2
PA	Proteome Analyst	Chapter 1.1, Chapter 2.5
QoS	Quality of Service	Chapter 2.4
RU	Resource Utilization	Chapter 2.3
SJF	Shortest Job First	Chapter 2.4
SPMD	Single Program Multiple Data	Chapter 6.3
TP	Throughput	Chapter 2.3
TREC	Transparent Result Caching	Chapter 6.2
VFS	Virtual File System	Chapter 5
VNM	Versioned Namespace Manager	Chapter 1.2, Chapter 3.2, Chapter 5
VNS	Versioned Namespace	Chapter 1.2, Chapter 3.3.1
WaFS	Workflow-aware File System	Chapter 1.2, Chapter 3.2, Chapter 5

Chapter 1

Introduction

1.1 Research Goal

Many high-performance computing (HPC) and scientific *workloads* (i.e., the set of computations to be completed), such as those in bioinformatics [91, 102], biomedical informatics [42], cheminformatics [82] and geoinformatics [67], consist of jobs with control-flow or dataflow dependencies, represented as a *Directed Acyclic Graph* (DAG). *Control-flow dependency* specifies that one job must be completed before other jobs can start. In contrast, *dataflow dependency* specifies that a job cannot start until all its input data (typically created by previously completed jobs) is available. Control-flow is the more commonly used abstraction to reason about the relationship between different jobs, but we show how dataflow information is also valuable.

Dataflow dependency signifies the *actual* information dependency— what compiler writers call *true dependency* [71]— requirements of the computation, whereas control dependencies may or may not imply information dependencies. Dataflow ensures the correctness of the computation under the assumption that there is no data-sharing via side effects (e.g., a shared read-write file). It should be noted that *not* all computations are easily characterized simply in terms of either their control-flow or dataflow. For example, a common database breaks the producer-consumer relationship of a DAG. However, many HPC workloads are of the form described here.

For example, the *Proteome Analyst* (PA) web service [91] has a multistage *Fork&Join* workflow (Figure 1.1) that classifies the proteome (i.e., all of the proteins of an organism, usually represented as a set of strings) in terms of its molecular function and subcellular localization. In this example two pipelines are constructed. One pipeline is `Job A, B, D` and `F`, and the other is `Job A, C, E` and `F`. In this case the shapes of the control-flow DAG and the dataflow DAG are the same, but, in

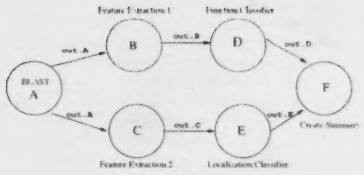


Figure 1.1. Proteome Analyst Workflow (a 6-job Fork&Join workflow)

general, they do not have to be the same.

In practice, a workload is often composed of multiple instances of the *same* workflow, with each instance acting on (possibly) independent input files or different initial parameters. In our example, analyzing a proteome may require one instance of the workflow (e.g., Fork&Join) for each protein in the proteome. Therefore, a proteome analyst workload would consist of multiple instances or copies of the workflow in Figure 1.1.

The goal of this thesis is to maximize the performance of these kinds of workloads in HPC systems. The primary metric of performance is *makespan*, which is the turnaround time to complete all jobs in a workload.

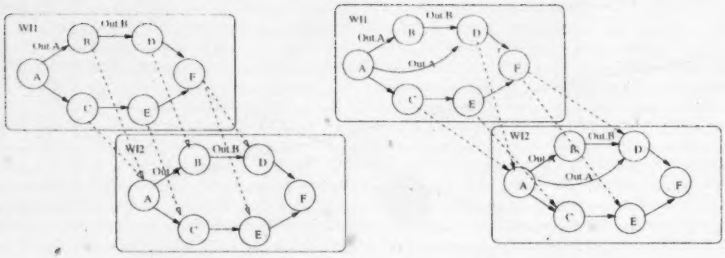
1.1.1 The Challenges

To achieve our goal, we have a key observation that from a control-flow perspective, workflow instances are inherently independent. However, in the context of a shared file system, where the namespace and finite resources are shared, interactions between instances can lead to incorrect executions. Whether the issue is *anti* or *output dependency* [71] on files or simple competition for storage resources, the selective isolation of each workflow instance can be important for maximizing scheduling flexibility and performance. However, in practice, realizing this benefit is not always straightforward due to a variety of problems and constraints:

1. *Filename conflicts:* the unmediated interaction of different workflow instances can lead to a problem of filename conflicts between concurrent workflow instances overwriting common files (i.e., one can erroneously overwrite each other's data).

   As with compiler optimizations, our solution is based on a renaming strategy. Unlike related work on scheduling, our renaming is automatically provided by the file system and scheduler (Chapter 3.2).

2. *Deadlocks:* the tens or hundreds of concurrent workflow instances can overwhelm any finite



(a) Control–flow–based (Unsafe) (b) Dataflow–based (Safe)

Figure 1.2. Control-flow is not always safe to use in exploiting inter-workflow instance concurrency. An example (a) shows how File Out.A is unsafely overwritten by Job A in WI2 before Job D in WI1 consumes it (The data dependency between Job A and Job D is not shown in the control-flow). The dataflow example (b) is correct in that both Job C and Job D in WI1 must complete before Job A in WI2 can overwrite File Out.A. Note that the dataflow from Job A to Job D is only specific to (b) for comparison purposes.

storage resource constraint and lead to *deadlock*.

Unlike related work on deadlock, which focuses on how deadlock can be prevented, avoided, or detected, we focus on deadlock avoidance while improving the *active* utilization of resources (Chapter 4.2 and 4.5) to improve makespan.

The batch schedulers in most current HPC systems (e.g., Condor [94], PBS [47] and LSF [108]) are control-flow driven (i.e., *control-flow-driven batch schedulers*; see Chapter 2.1) in the sense that job scheduling is generally based on the inter-job control dependencies (i.e., control-flow) specified by users. However, such control dependency information tends to be insufficient to achieve better job scheduling in terms of high performance and low storage overhead given the problems or constraints identified above.

To resolve the filename conflicts, most current batch schedulers adopt a *sub-directory-based strategy* (also called *Sub-dir* in the later discussion) that creates a *working directory* for each workflow instance and moves all required data to that directory (e.g., GEL [64], Triana [93] and DAGMan [18]). Without any dataflow knowledge of what files are used within the workflow instance, control-flow-driven batch schedulers have little choice but to partition the file namespace in a brute-force renaming strategy.

Specifically, all the computations of the instance are carried out in that directory. However, based on the control-flow information alone, it is not always possible to determine when a file

will no longer be used by the other jobs. For example, in Figure 1.2(a), after `Job B` in workflow instance `WI1` finishes, `Out.A` cannot be deleted immediately because we do not know if it will be used by other jobs such as `Job D`, `Job E` or even `Job F`. Therefore, files are usually not deleted immediately even when they can be deleted, thereby incurring potentially large storage overhead, especially when a large number of workflow instances execute concurrently and access large files.

To maximize the concurrency while minimizing the storage overhead, *overwrite* is another strategy that is often used. However, a control-flow-driven batch scheduler cannot always ensure the correct overlap of multiple workflow instances with respect to safe file overwriting. A different example in Figure 1.2(a) illustrates why `Job A` in workflow instance `WI2` cannot be safely overlapped with `Job D` in `WI1`. The control-flow-based overwrite strategy might assume that since `Job B` and `Job C` in `WI1` are finished, the output of `Job A` (i.e., `Out.A`) in `WI1` can be overwritten. However, the overwrite may be premature, leading to an incorrect schedule. Thus, in practice, a common solution is to execute each workflow instance in a sequential order (also called *BASE* policy in the later discussion). Although this serial policy is simple and incurs small storage overhead, it does not allow any *inter-instance concurrency* (Chapter 2.1) and thus suffers from poor performance.

Traditionally, in most current HPC systems, the batch scheduler dispatches jobs without any concern about the interaction between jobs and the underlying file system. Therefore, when storage is limited, such non-coordinated allocation among jobs may cause the system to enter a deadlock state due to jobs waiting to allocate file space.

However, the control-flow-driven batch scheduler, even with the knowledge of the storage requests of each job, cannot *effectively* resolve the deadlock problem. On one hand, as discussed, the files that are no longer useful cannot be deleted immediately and thus *all* the intermediate files inside a workflow instance have to be kept until *all* the jobs in the instance are finished, thereby increasing the storage overhead. We can see a scenario from Figure 1.2(a) where `Out.A` (or any other file, if only control-flow is known) cannot be deleted until instance `WI1` finishes. But if `Out.A` is only accessed by `Job B`, and the file could potentially be deleted after `Job B` completes, but the control-flow graph does not capture that information. Therefore, to be safe, all files can only be deleted when the entire instance completes.

On the other hand, based on the control dependencies between the jobs, the *maximum resource demand or claim* (*maximum claim* for short) associated with each instance in general is hard to accurately approximate since each intermediate file is assumed to be used until the end of the instance. The concept of having a good estimate (i.e., a tight upper bound) for the maximum claim is also central to the banker's algorithm [43]. But without reliable user- or system-provided information,

the global maximum claim must usually be taken to be the conservative sum of *all* the requests of all the jobs in the instance, which is, in practice, a loose upper bound on the maximum actual resource requests. Unfortunately, loose upper bounds on the maximum claim can lead to poor resource utilization (Chapter 4.5). Furthermore, global and static maximum claims may lead to poor active utilization of resources as the local and dynamic behavior of the workflow instances changes (Chapter 4.5).

Therefore, to resolve the deadlock, traditional batch schedulers usually delegate the responsibility to users who have to manually intervene after deadlock has been detected, imposing the problem-solving efforts on the users, and resulting in lost progress due to the need to stop and re-run jobs.

1.1.2 The Advantages of Dataflow Information

To address these problems, we argue that having the dataflow information is fundamentally advantageous to determining the precise scope and time window when resources are required. Some advantages can be observed in the following scenarios:

1. **Solving the Resource Deallocation/Prompt Release Problem:** In Figure 1.2(a), given the dataflow information, we know that the file `Out.A` in `WI1` can be deleted immediately after `Job B` finishes because no further jobs will need that file.

2. **Solving the Premature Resource Release/Re-Use Problem:** In Figure 1.2(b), with the dataflow information, we know it is necessary to delay the start of `Job A` in `WI2` until after `Job C` and `Job D` in `WI1` are completed. Delaying the start of `Job A` maintains correctness without requiring any additional resources for a file renaming strategy. And since `Job A` of instance `WI2` can still overlap `Job F` of `WI1`, there is still inter-workflow instance concurrency.

3. **Solving the Poor Resource Requirement Estimation Problem:** Of course, dataflow is not essential to deadlock resolutions, but it is useful to design algorithms with better performance. To improve efficient storage utilization, the dataflow information can be exploited at runtime to compute more accurate *localized maximum claims* for each instance as opposed to a pre-computed global maximum claim in the control-flow-driven batch scheduler. For example, in Scenario 1 (Resource Deallocation Problem) above, the localized maximum claim associated with `Job D` in `WI1` can be computed as the sum of all jobs' requests deducted by the size of `Out.A` since we know, based on the dataflow information, that after `Job B` finishes, `Out.A` can be deleted immediately.

In this study we demonstrate the value of dataflow information in workflow batch scheduling, with a focus on maximizing job concurrency given the filename conflicts and reducing the impact of deadlock when storage resources are constrained.

1.2 Contributions

Although some ad hoc solutions exist and other systems have attempted to address these problems, we are advocating a more systematic and comprehensive solution. More specifically, our contributions are the following:

1. **New Policies Exploiting Dataflow to Maximize Concurrency** (Chapter 3):

   We introduce three new dataflow-based scheduling policies to maximize concurrency (and minimize makespan) by reducing the impact of the filename conflict and deadlock problems.

   (a) *Versioned Namespace* (VNS)

   (b) *Overwrite-Safe Concurrency* (OSC)

   (c) *Hybrid Policy* (HB), the combination of VNS and OSC, including the proposed deadlock avoidance algorithms in Contribution 2 below.

   Both VNS and OSC take advantage of dataflow information to maximize the inter-workflow instance concurrency. HB, the combination of VNS and OSC, can trade off performance for the storage overhead, and thus it is much more flexible than the other two policies. In this study the various approaches to deadlock avoidance are in the HB policy.

   Main quantitative evidence/results (Chapter 3.4): Both OSC and VNS are shown to reduce makespans, relative to BASE (the baseline in reality), and reduce storage overhead, relative to Sub-dir (the common practice in reality). These results demonstrate that dataflow information is valuable in addressing the filename conflict problem by improving job concurrency while minimizing storage overhead.

2. **Dataflow Information Improves Active Resource Utilization with Deadlock Avoidance** (Chapter 4): We integrate two novel concepts with the traditional problem of deadlock avoidance. First, we show how knowledge of dataflow information can be exploited at runtime to compute localized maximum claims and reduce makespan when deadlock is a potential problem. Second, we show how a distinction between *Active*, *Inactive* and *Free* resources, as opposed to just *Allocated* versus *Unallocated* resources, is important to minimizing makespan.

Here, the active resources refer to the allocated resources that are held by the running jobs, whereas the inactive resources are also allocated but are held by the blocked jobs due to resource constraints. The remainder are free (i.e., unallocated) storage.

First, with *Dataflow-based Aggregate Requests* (DAR), the maximum claim of each instance is dynamically computed by summing the resource requirements of all the remaining jobs (i.e., those jobs that have not yet been finished), instead of using a static pre-defined value. Second, the *Dataflow-based Topological Ordering* (DTO) algorithm exploits the dataflow knowledge to topologically order the jobs in the current instance when checking for safety (i.e., a specific order of job completion that is within a resource budget). Both algorithms try to maximize the active storage utilization by either improving the inter-instance concurrency or improving the *intra-instance concurrency* (Chapter 2.1).

Main quantitative evidence/results (Chapter 4.5): DAR and DTO outperform the banker's algorithm and Lang's algorithm [60] with respect to makespan and active storage utilization for the workloads with a variety of workflow shapes, workflow sizes and other workflow parameters. In addition, we show that, unexpectedly, Lang's improvements to the banker's algorithm do not always result in improved makespans. As designed, Lang's algorithm *does* improve total storage utilization, but much of the utilized storage is inactive utilization, which does not improve makespan. This result shows that making a distinction between active and inactive storage is important to minimizing makespan.

3. **WaFS for Dataflow Collection** (Chapter 3 and Chapter 5): We propose and prototype a novel system called *Workflow-aware File System* (WaFS) that extends a traditional file system to provide a *distinct namespace* for each workflow instance (to address the filename conflict problem) and transparently gather the dataflow information to help the scheduler. Unfortunately, the dataflow information is not usually available from the user submission in control-flow-based systems or tracked by the traditional file systems.

   To overcome these challenges, we show how an enhanced *Versioned Namespace Manager* (VNM) can be layered on top of a traditional file system to integrate the file system and the batch scheduler as a *WaFS Scheduler*. The WaFS Scheduler uses WaFS to collect dataflow information and stores that dataflow information in the VNM, and the modified scheduler exploits the dataflow information for better scheduling.

   WaFS is primarily a proof-of-concept prototype of a new combined file system and scheduler architecture. A full evaluation of different implementation strategies is beyond the scope of

this dissertation.

In summary, we characterize the proposed policies and the associated deadlock avoidance algorithms in Table 1.1. For comparison purposes, some reference policies and algorithms are also listed. In the table, Batch Scheduling is characterized by three features: *Concurrency*, *Storage Allocation Granularity* and *Dataflow Collection*. Concurrency is related to the job scheduling, which can be obtained from intra-instance concurrency and inter-instance concurrency (see Chapter 2). Both kinds of concurrency are limited by dataflow information or control-flow information, or (total/active) storage. Storage Allocation Granularity refers to the computation unit (instance or job) by which the storage is allocated. Dataflow Collection is only available to dataflow-based batch scheduling. Deadlock Avoidance Algorithms are distinguished by the computation of maximum storage demand/claim and how the safety check (i.e., the safety check of a job's request) is conducted. Maximum resource demand/claim can be characterized by three features: what type of graph is used in its computation, dataflow or control-flow, whether the scope of the computed value is local or global to the associated instance, and whether the maximum claim is computed, static or dynamic. Safety checking refers to what kind of safe sequence, either instance or job, is constructed when a request is made.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 provides background information, including some important concepts that are directly relevant to this thesis. Workflow-aware File System (WaFS) for dataflow collection and scheduling policies, together with their performance evaluations, are presented in Chapter 3. The deadlock avoidance algorithms for the hybrid policy and their evaluation results are presented in Chapter 4. Chapter 5 describes a simple prototype of WaFS. Chapter 6 covers some related work. The conclusions are summarized in the last chapter.

Policy Name	Batch Scheduling (Chp. 1, 3, 4.3, and 5)				Deadlock Avoidance (Chp. 4)			
	Concurrency (Chp. 3.3 and 4.3) (Job Scheduling)		Storage Allocation Granularity (Chp. 3.3 and 4.3)	Dataflow Collection (Chp. 3.2 and 5)	Maximum Storage Demand/Claim & Safety Checking (Graph type, Scope, Time, Safe Sequence)			
	Intra-Instance limited by	Inter-Instance limited by			DAR (Chp. 4.2.1)	DTO (Chp. 4.2.2)	Banker's [43]	Lang's [60] (Appendix A)
BASE/Serial (Chp. 1)	Control-flow	Control-flow[a]	Instance	BASE and Sub-dir evaluated with unlimited resources (Chp. 3.4)				
Sub-dir (Chp. 1)	Control-flow	Total Storage	Instance					
VNS (Chp. 3.3.1)	Dataflow	Total Storage	Job	Available	VNS and OSC evaluated with unlimited resources (Chp. 3.4)			
OSC (Chp. 3.3.2)	Dataflow	Dataflow[b]	Job	Available				
HB (Chp. 3.3.3)	Dataflow	Active Storage (Chp.1, 4.2, and 4.5)	Job	Available	Dataflow (schedule independent), Local scope, Dynamic, Instance seq. (Chp. 4.5)	Dataflow (schedule dependent), Local scope, Dynamic, Job seq.[c] (Chp. 4.5)	Control-flow (schedule independent), Global scope, Static, Instance seq. (Chp. 4.5)	Dataflow (Pipeline) (schedule dependent), Local scope, Static, Instance seq. (Chp. 4.5)

Table 1.1. The Characteristics of the Proposed Policies and Associated Deadlock Avoidance Algorithms: BASE/Serial and Sub-dir correspond to our discussed serial policy and the policy that employs the sub-directory-based strategy to address the filename conflict problem. VNS, OSC and HB are our proposed policies. The HB policy contains several deadlock avoidance algorithms among which DAR and DTO are our proposed deadlock avoidance algorithms. Banker's and Lang's are reference algorithms for evaluating both DAR and DTO in the later chapters.

[a] possible filename conflict
[b] no filename conflict
[c] The jobs local to the instance making requests

Chapter 2

Background Knowledge

A survey of related work is given in Chapter 6. This chapter covers the related background knowledge (with relevant citations) for the main parts of the thesis.

2.1 Basic Concepts

A *job* is the execution of code by the system, such as a shell script, an interpreted program or a compiled application. Users submit jobs to the system, which in turn queues (if necessary), places and schedules the jobs. In a computation, a job is usually associated with some resource requests (e.g., the number of required processors, the estimated time to be executed, input/output (I/O) file sizes, startup parameters), which must be satisfied for the job execution to take place.

A *workflow* consists of a set of jobs with dependency relationships. With *control dependency*, some jobs must finish before other jobs can start. The control dependencies of a workflow can be represented as a *Directed Acyclic Graph* (DAG) whose nodes represent the jobs and whose edges denote their control dependencies. We call this graph a *control-flow graph* (CFG), denoted as $G_c = G(N, E_c)$. Figure 2.1 represents a *Fork&Join* workflow; its input script in Condor DAGMan [18], a well-known meta-scheduler for Condor jobs [94], is shown in Figure 2.2. This workflow consists of six jobs (`Job A`, `Job B`, `Job C`, `Job D`, `Job E` and `Job F`), and their control dependencies are described at the bottom of this script. The implicit node in Figure 2.1 does not represent a real job. It is a *virtual* job implied by the semantics of DAGMan to synchronize the executions of `Job B`, `Job C` and `Job D`, `Job E`.

A *workflow instance* is a concrete execution of the workflow with its own input data or parameters for each constituent job. It is created after the workflow is submitted to the scheduler. For example,

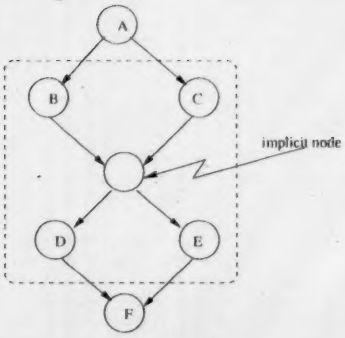


Figure 2.1. An Example of a Fork&Join Workflow

```
/* Filename: Fork&Join.dag */
Job   A   A.condor
Job   B   B.condor
Job   C   C.condor
Job   D   D.condor
Job   E   E.condor
Job   F   F.condor
Script   PRE   A top_pre.csh
Script   PRE   B mid_pre.perl $JOB
Script   POST  B mid_post.perl $JOB $RETURN
Script   PRE   C mid_pre.perl $JOB
Script   POST  C mid_post.perl $JOB $RETURN
Script   PRE   D mid_pre.perl $JOB
Script   POST  D mid_post.perl $JOB $RETURN
Script   PRE   E mid_pre.perl $JOB
Script   POST  E mid_post.perl $JOB $RETURN
Script   PRE   F bot_pre.csh
PARENT   A   CHILD B C
PARENT   B C   CHILD   D E
PARENT   D E   CHILD   F
```

Figure 2.2. An Example Input Script File for DAGMan

the user can submit the input script file in Figure 2.2 to DAGMan to create a workflow instance. Each submission, possibly of the same input script, creates a different workflow instance.

Assuming no job preemption, a job progresses through different states:

- **Submitted State:** A job in this state is generally blocked unless its control dependencies are already resolved. The time when a job enters this state is usually the job's submission time, denoted as *job.submitTime*.
- **Ready State:** Whenever a job's control dependencies are resolved (all its parent jobs have completed), it enters the ready state, waiting to be scheduled. The time when it enters the ready queue is defined as the ready time of the job and denoted as *job.readyTime*.
- **Running State:** When resources become available and a job is selected by the scheduler to be executed, the job enters the running state. The time when it starts running is the job's start time, denoted as *job.startTime*.
- **Done State:** when a job finishes its computation, it enters the done state. The time when it enters the done state is denoted as *job.doneTime*, which is the job's completion time.

In our discussion, a workload consists of a set of workflow instances that come from the *same* workflow. We characterize a workload from several aspects. The first is the *shape* of the workflow,

such as Pipeline, Fork&Join and Lattice. The shape of a workflow determines the inherent degree of job concurrency within each workflow instance. The second is the *average inter-arrival time* of the workflow instances, which is the difference in *job.submitTime* between different jobs and characterizes the arrival rate of workload submission (i.e., offered load). The third is the *service time* of the jobs within each workflow instance; the sum of them in a workload reflects the total computational work. In addition, when storage resources are limited and deadlock is a pragmatic concern, other characteristics of the workload such as the *workflow size* and *file sizes* are also important to the performance goal(s) (this will be discussed in the next section).

Job scheduling [29, 84] is the process of computing a plan (or schedule) that maps each ready job to a processor or a computing host to achieve the performance goal(s). This process is usually accomplished by following the constraints and precedences specified in the control-flow graph (Figure 2.2). Unlike control dependency, *data dependency* refers to a producer-consumer relationship between two or more jobs where a job cannot start until all its input data (which is typically generated by the previously completed jobs) is available. Similar to control dependencies, data dependencies of a workflow can also be represented as a DAG. We call it a *dataflow graph* (DFG), which can be denoted as $G_d = G(N, E_d)$, where the edges, E_d, reflect the data dependencies between the nodes, N.

Thus, under the assumption that there are only file-based dependencies and no side-effects with globally shared data, the dataflow graph captures the fundamental dependencies among jobs, which must be respected to ensure the correct computation. In contrast, the dependencies in the control-flow graph are generally created for the convenience of the programmer. For example, in Figure 2.2, the line

PARENT `B C` **CHILD** `D E`

produces four control-flow dependencies (see the sub-graph surrounded by the dashed line in Figure 2.1):

```
B to D
B to E
C to D
C to E
```

even though, in fact, there are no data dependencies from `Job E` to `Job B` and from `Job D` to `Job C`. DAGMan introduces an implicit synchronization point between `Job B`, `Job C` and `Job D`, `Job E` for the users to easily reason about their workflows, but the additional synchronization

Global Variable	Size	Stored Data
r	1	$r(t)$ is the number of available resources at the moment t
max_claim	n	max_claim(i) is the maximum resource claim that process i will request
alloc	n	$alloc(i,t)$ is the number of resources that have been allocated to process i before the moment t
need	n	$need(i,t) = max\_claim(i) - alloc(i,t)$, i.e., the max number of resources that process i still needs to complete its task at the moment t

Table 2.1. Some Global Variables Used in the Banker's Algorithm: n is the number of processes that are involved in deadlock avoidance.

results in (potentially) lower concurrency than using the corresponding dataflow graph. As for the benefits of dataflow, we will give more details in the later sections.

Based on the above definitions, we make a further assumption that the shape of a workflow's dataflow graph is independent of the workflow's input data and job parameters (i.e., static dataflow graph). Although this assumption seems restrictive, we believe it is quite reasonable in scientific computation.

A *Control-flow-driven batch scheduler* schedules jobs based on G_c, which is usually specified by users. Most current batch schedulers are generally identified to be control-flow-driven. In contrast, the *Dataflow-driven batch scheduler* schedules jobs based on G_d. G_d is not always available to a control-flow-driven batch scheduler, or even if it is available, it might not be used by a control-flow-driven batch scheduler. For example, in the DAGMan script shown in Figure 2.2, the *PRE* and/or *POST* sub-scripts are commonly used to stage in and/or stage out files in some area for the cluster jobs. These sub-scripts actually specify the dataflow information, but such information is not used in the DAGMan scheduler.

2.2 Deadlock and the Banker's Algorithm

When multiple workflow instances run concurrently and compete for the limited storage resources, deadlock can occur. Generally, there are three approaches to dealing with deadlock, *deadlock prevention*, *deadlock avoidance* and *deadlock detection combined with recovery* [92], each with advantages and disadvantages with respect to the resource utilization and computation overhead.

The banker's algorithm [23] is the most widely recognized deadlock avoidance algorithm. The basic idea of this algorithm to prevent deadlock is to deny or postpone the request if granting the request could put the system in an unsafe state (one where deadlock could occur). The pseudo-code

```
/* Process P_i makes a resource request b_i */
banker(P_i, b_i) {
    if (b_i > r(t))
        /* wait because there aren't enough free storage*/
        return false;
    else
        /* pretend to modify the system */
        r(t) ← r(t) − b_i;
        alloc(i,t) ← alloc(i,t) + b_i;
        need(i,t) ← need(i,t) − b_i;
        if (safetycheck(P_i))
            return true;
        else
            /* undo the changes since allocation could course
            ** deadlock and try the request later once resources
            ** have been cleaned up. */
            r(t) ← r(t) + b_i;
            alloc(i,t) ← alloc(i,t) − b_i;
            need(i,t) ← need(i,t) + b_i;
            return false;
}
```

Figure 2.3. The Banker's Algorithm for Requesting a Resource Allocation

for the banker's algorithm for a single type of resource is shown in Figure 2.3 and 2.4.

Although the banker's algorithm has more potential to improve resource utilization due to its dynamic safety check, it has a basic premise that the maximum amount of resources required by each process need to be declared a priori. Even with this premise, the banker's algorithm still forms the basis for many deadlock avoidance algorithms in a variety of application contexts [8, 10, 60, 61].

2.3 Performance Metrics

The execution of a workflow G may use different numbers of computational nodes at different time periods. For each time period, the number of computational nodes used to execute a workload is defined as the *Degree of Concurrency* (DOC). We can use DOC to measure the instantaneous concurrency of a workflow instance. Note that DOC_{G_c} (i.e., DOC based on G_c) may be different from DOC_{G_d} (i.e., DOC based on G_d). Based on this concept, we can further compute *Average DOC*:

$$Avg.\ DOC = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} DOC_G(t)dt \tag{2.1}$$

to measure the aggregate concurrency (t_1 and t_2 are two time points).

When multiple instances of the same workflow execute concurrently, the concurrency may come

```
/* checking if system is in a safe state at moment t.*/
safetycheck(P_p) {
    /* process[ ] is an array of processes to
    ** record if a process can be finished
    ** in the safety check. process[ ]
    ** and resource are local variables. */
    resource ← r(t); /* available resource */
    for (each P_i in the process[ ]) do
        process[i] ← false;
    while (∃P_i such that process[i]
        = false ∧ need(i,t) ≤ resource) do
        resource ← resource + alloc(i,t);
        resource[i] ← true;
        if (i = p)
            return true; /* state is safe */
    return false; /* state is unsafe */
}
```

Figure 2.4. The Safety Checking Algorithm in the Banker's: The algorithm is performed to find out if the system is in a safe state.

from two aspects. One is called *intra-workflow instance concurrency* (*intra-instance concurrency*, for short), referring to the number of concurrent jobs that belong to the same workflow instance. The other is called *inter-workflow instance concurrency* (*inter-instance concurrency*, for short), referring to the number of concurrent instances. Both intra- and inter-workflow instance concurrencies are vital for the scheduler to make the best use of system resources and hence achieve high-performance. The following metrics are frequently used in measuring the performance of different scheduling algorithms on a set of jobs, denoted as *Jobs*:

- Makespan (MS):

$$MS = \max_{job \in Jobs} job.endTime - \min_{job \in Jobs} job.submitTime \tag{2.2}$$

  Intuitively, makespan is the amount of time the system takes to complete *all* of the jobs of a workload, from the submission of the first job to the completion of the last job.

- Mean Response Time (MRT):

$$MRT = \frac{\sum_{job \in Jobs} job.responseTime}{|Jobs|} \tag{2.3}$$

  where $job.responseTime = job.endTime - job.submitTime$ and $|Jobs|$ denotes the num-

ber of jobs in the job set, *Jobs*. Mean response time is the length of time the system takes, on average, to complete a job after it has been submitted.

- Throughput (TP):

$$TP = \frac{|Jobs|}{MS} \tag{2.4}$$

Throughput represents a metric to show how many jobs can be finished during a given time unit.

Primarily, users are concerned with both makespan and mean response time as they are most often interested in minimizing the total computation time and the delay between job submission and job completion. In contrast, system administrators usually care about throughput as the throughput generally reflects the overall performance of the system. However, in our context, makespan is more important since the workload as a whole is usually the concern to carry out a well defined computation task (i.e., a parameter-based study) rather than each individual job or instance.

- Resource Utilization (RU):

$$RU = \frac{\sum_{job \in Jobs} \int_0^{MS} job.reqRes(t)dt}{MS \cdot N} \tag{2.5}$$

where N is the total number of available resources of the system (e.g., nodes of the cluster, storage units), and $job.reqRes(t)$ = the number of requested resources by the job at the moment t. Utilization refers to the percentage of the time that resources are busy or occupied with useful work. Sometimes this metric is also useful in the evaluation of the scheduling performance.

2.4 Scheduling in Batch Queuing Systems

Scheduling algorithms on supercomputers or clusters can be broadly classified into two categories: *time-sharing* and *space-sharing*. Time-sharing algorithms divide time on a processor into discrete intervals or *time slices* and then assign these time slices to unique jobs. As a result, multiple jobs can share the same computing resources by preempting jobs and alternating between different jobs across different time slices. Conversely, space-sharing algorithms partition the processors into disjoint sets and execute each parallel job in a distinct partition until the job completes. Batch

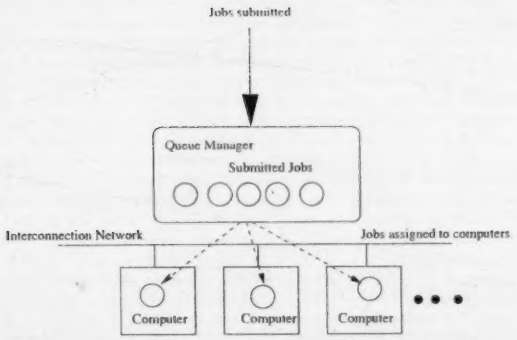


Figure 2.5. An Architecture of a Typical Batch Queuing System: A central job queue is maintained by Queue Manager, which performs some particular scheduling algorithm to map jobs to a set of interconnected computers.

queuing systems are generally based on space-sharing algorithms.

A batch queuing system is often used as a resource manager for a supercomputer or a cluster. The purposes of the queuing system are to maximize the utilization of shared resources and to fairly share the resources between users. Figure 2.5 shows an architecture of a typical batch queuing system. Jobs are submitted to a centralized queue master where a scheduling algorithm is performed to map each job to the assigned computer(s).

In batch queuing systems, some typical space-sharing algorithms used are *First Come First Serve* (FCFS) and *Shortest Job First* (SJF). FCFS schedules jobs according to the order they enter the queue. This algorithm is simple and easy to implement. It can also produce a fair and predictable schedule. But FCFS's mean response time is may be high when jobs with long service times arrive before short jobs if many shorter jobs continue to arrive.

SJF addresses this problem by periodically sorting the incoming jobs according to their service time and scheduling the shortest job first, thereby lowering the mean response time. However, unlike FCFS, SJF requires the user to estimate the job service time. In its pure form, SJF can also lead to starvation for long jobs.

To fulfill user requirements and improve system performance, in practice these basic scheduling algorithms are enhanced by integrating a variety of new functionalities. Some of them are as follows:

- **Fair-share Strategy:** This strategy refers to the ability of a batch scheduler to treat each user fairly in terms of resource allocation when a system is heavily loaded. One of the methods to achieve this ability is to allow the batch scheduler to dynamically adjust the priorities of the

jobs in the queue based on the historical data of previously executed jobs.

- **Advanced Reservation:** Advanced reservation is a technique used by batch schedulers to ensure the QoS for applications. This functionality is usually achieved by using the user-estimated execution time to reserve system resources such as processors and memory in advance and thereby computing a qualified schedule. For example, a reservation may allow the user to start an arbitrary number of interactive or batch jobs during the reserved time frame. Moreover, deadline scheduling can be implemented to guarantee that a batch job with a deadline notification is completed at (or before) the specified time.

- **Backfilling Technique:** Backfilling is a technique used to improve the resource utilization (especially of the processors) in space-sharing scheduling algorithms [30, 37, 65]. Backfilling tries to improve system utilization by allowing the jobs with low priorities to *bypass*[1] those with high priorities so that the otherwise idle resources can be utilized. Currently, many batch queuing systems (e.g., LoadLeveler [53], LSF [108] and PBS [47]) have implemented this technique.

- **Job Dependency:** Many existing batch queuing systems (e.g., PBS [47], LSF [108], CODINE [39], Condor [94] and LoadLeveler [53]) allow the user to specify job dependencies (i.e., control dependencies). Since creating an optimal schedule for a set of dependent jobs under space-sharing scheduling is generally NP-complete [58], heuristic algorithms based on some well-known algorithms (e.g., *List Schedules* [1] and *Clustering Algorithms* [35]) are generally used.

- **Resource Constraints:** Scheduling with resource constraints allows users or batch queuing systems to enforce limits on multiple resources so that resource over-subscription on the system can be prevented. For example, in NQE [21] the memory limits per system and per batch queue can be defined in advance, and in PBS [47] multiple batch queues can be defined with hard limits on a number of resources (e.g., memory, storage space and CPU time) available to each queue. When considering job dependencies, resource constraints might cause deadlock due to resource competition. However, few existing batch systems address it at the system level. Rather, in general, they delegate the responsibilities to the users.

[1] The computed schedules of the jobs with high priorities are not altered.

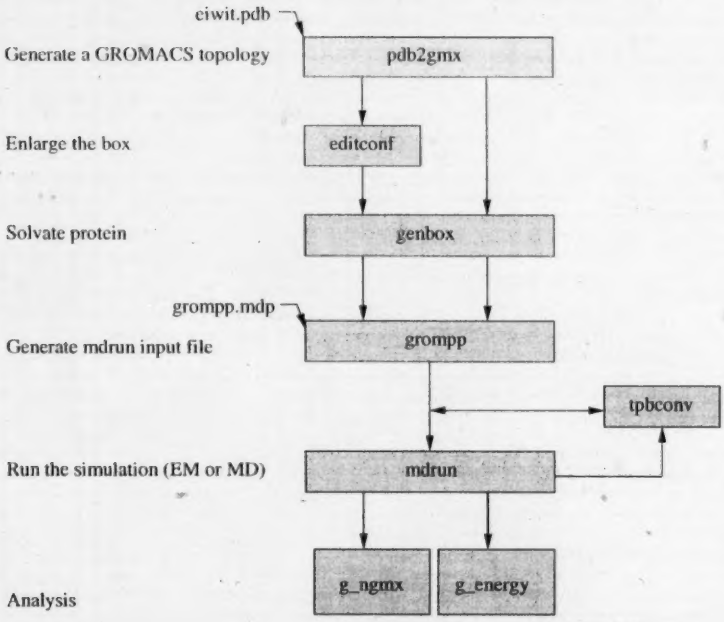


Figure 2.6. An Example of Typical GROMACS Workflow Chart

2.5 Some Typical Workflow Applications

We describe three typical workflow applications [38, 42, 91] (Table 2.2), which have different workflow shapes and are representative of many scientific computations.

2.5.1 GROMACS: Molecular Dynamics

Molecular Dynamics (MD) is a computer simulation that helps people to " understand the properties of assemblies of molecules in terms of their structure and the microscopic interactions between them" [3].

"[MD] acts as a bridge between microscopic length and time scales and the macroscopic world of the laboratory: we provide a guess at the interactions between molecules, and obtain 'exact' predictions of bulk properties. The predictions are 'exact' in the sense that they can be made as accurate as we like, subject to the limitations imposed by our computer budget. At the same time, the hidden detail behind bulk measurements can be revealed" [3].

The simulation consists of the numerical, step-by-step, solution of the classical equations of motion (e.g., the Newtonian equations of motion for systems with hundreds to millions of particles).

Application	Functions	Workflow Shape
GROMACS [42]	A molecular dynamics simulation package to simulate the Newtonian equations of motion for systems with hundreds to millions of particles	Pipeline
Proteome Analyst (PA) [91]	A bioinformatics tool to predict protein properties such as the general function and the subcellular localization of proteins using machine learning techniques	Fork&Join
Bronze Standard Medical Imaging (BSMI) [38]	A data intensive medical image processing application developed to overcome the difficulties of evaluating the accuracy and robustness of image processing algorithms when the reference image is not available	Lattice

Table 2.2. Three Typical Workflow Applications: GROMACS, Proteome Analyst (PA) and Bronze Standard Medical Imaging (BSMI)

GROMACS [42] is a versatile collection of programs and libraries to perform the molecular dynamics and the subsequent analysis of the trajectory data.

Typically, a GROMACS workflow has a Pipeline shape. Figure 2.6 shows a typical GROMACS MD run of a protein in a box of water. Several steps of energy minimization may be necessary; these consist of cycles: $grompp \rightarrow mdrun$. The details of each stage are reported in [42].

2.5.2 Proteome Analyst (PA): Bioinformatics, Machine Learning

Proteome Analyst (PA) [91] is a bioinformatics tool developed at the University of Alberta to predict protein properties such as general function (i.e., what does the protein do) and subcellular localization (i.e., where in the cell does the protein perform its main function) using established machine learning techniques.

The basic PA workflow has a Fork&Join shape (see Figure 2.7). It first accepts a *proteome* (i.e., a blend of proteins and genome that is often used to describe the entire complement of proteins expressed by a genome, cell, tissue or organism) in the form of a text string, and then uses *BLAST* [4] to find the *homologs* among known proteins for each given protein. During this process PA also gains information about InterPro [2] families, which can also provide information about homology. PA uses this information to predict the classes of proteins. More specifically, the feature extraction programs (i.e., *Feature Extractions* in Figure 2.7) take the homologs as the input and use different algorithms to extract some keywords or annotations as features. The extracted features are classified by different trained *classifiers* to determine the function and the localization for each query sequence

[2] InterPro is an integrated documentation resource for protein families, domains and functional sites.

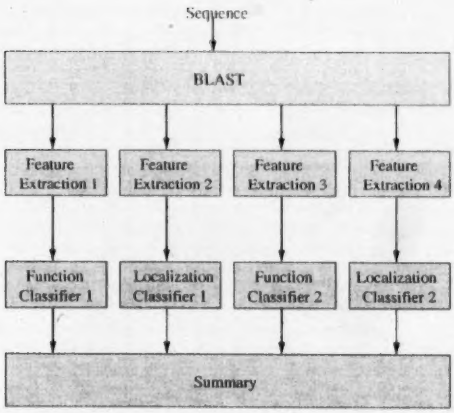


Figure 2.7. An Example of a Typical PA Workflow Chart

within the cell. Finally, the program *Summary* gathers, summarizes and presents the outputs from various classifiers.

In practice, as a minor simplification, if the proteome has 1000 sequences, then there will be 1000 workflow instances in the workload. The large number of instances make the PA workload an HPC and scheduling problem.

2.5.3 Bronze Standard Medical Imaging (BSMI): Medical Image Processing

Bronze Standard Medical Imaging (BSMI) [38] is a data-intensive medical image processing application developed to overcome the difficulties of evaluating the accuracy and robustness of image processing algorithms when the *ground truth* (i.e., reference image) is not available. The bronze standard method used in the application does not require the ground truth. Instead, it estimates the ground truth by leveraging the redundant information in all possible registered pairs of images.

The application is a workflow assembled from a set of basic tools (i.e., jobs, see Figure 2.8(a)), each having its own function to process the data, extract quantitative information and analyze results. The workflow can be simplified to be a Lattice-like workflow shown in Figure 2.8(b),[3] in which four major different registration algorithms [73] are used:

1. *Baladin:* an intensity-based algorithm that uses a block matching strategy to extract feature points in two images (e.g., the reference and the floating images) to be registered.

[3]The shape of BSMI workflow can be transformed into a Lattice-shaped workflow. This simplification is common in literature to abstract the real workflow [15, 69, 76].

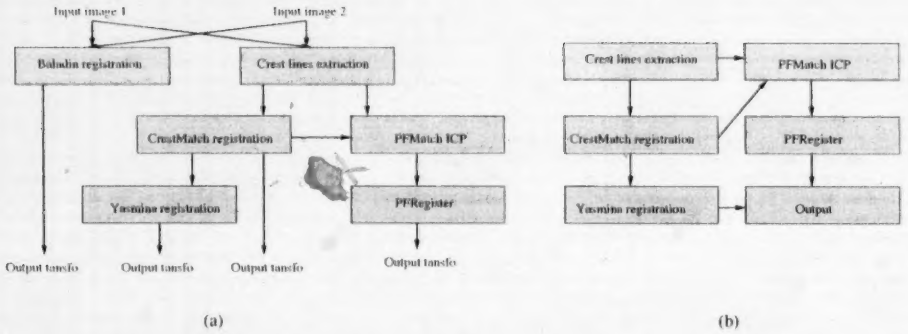


Figure 2.8. An Example of a Bronze Standard Medical Imaging Workflow Chart (a). We view it as a Lattice-like workflow shape (b) in our discussion.

2. *Yasmina:* an intensity-based algorithm built on top of the Powel algorithm to optimize a similarity measure on the complete images.

3. *CrestMatch:* a prediction-verification method used to initialize all the other algorithms to ensure that all algorithms converge toward the same minimum.

4. *PFRegister:* an Iterative-Closet-Point (ICP)-based algorithm to register the features (extracted from the input images) with more complex structures than points.

In addition to computation, these algorithms are also responsible for data exchange. In practice, a BSMI workload usually needs to process hundreds of individual medical images; thus, data parallelism is desired. The data parallelism is achieved through concurrent execution of multiple instances of the BSMI workflow.

2.6 Concluding Remarks

This chapter covered the background knowledge that is directly relevant to this thesis. First, we introduced some important concepts in workflow scheduling, which are useful to understand our *WaFS Scheduler*, including its mechanisms and policies for collecting and exploiting dataflow information. In addition, we discussed some performance metrics that are often used in the evaluation of batch scheduling algorithms. Among the discussed metrics, makespan and average degree of concurrency (i.e., Average DOC) are our major concerns. Finally, we described three typical workflow applications in molecular dynamics, bioinformatics and medical image processing. These applications have different workflow shapes (i.e., Pipeline, Fork&Join and Lattice) and are representative

of many scientific computations. They motivate us to use Pipeline, Fork&Join and Lattice workflow shapes in evaluating our scheduling policies and algorithms (Chapter 3.4 and Chapter 4.5).

Chapter 3

Dataflow Collection and Scheduling Policies

In this chapter we introduce our dataflow-based scheduling policies and how they exploit the *Workflow-aware File System* (WaFS) implementation (discussed in Chapter 5). We discuss the policies before the implementation because our implementation (Chapter 5) is simply a prototype, and any implementation or architecture that reliably gathers dataflow information can be used with our policies. In other words, our policies are more general than any specific implementation.

Our dataflow-based scheduling policies rely on having a mechanism to collect the dataflow information for batch scheduled jobs (Chapter 3.2, Chapter 5). Then they exploit the information to maximize job concurrency within the workflows despite possible filename conflicts and deadlock. Knowing the *true dependencies* between the jobs [71] (i.e., the dataflow among jobs) enables a file renaming strategy that eliminates artificial bottlenecks to concurrency while efficiently using resources. Through a simulation-based study we show the potential benefits of the use of dataflow information to job concurrency and the trade-offs that can be made between storage overhead and performance. Note that, unlike later chapters that consider deadlock, in this chapter we assume that there are an arbitrary number of processors and storage resources. In other words, this chapter studies limits to concurrency other than simple resource limits (e.g., data hazards [71]).

3.1 Motivation

There are two major challenges to collecting and using dataflow information:

1. **Dataflow information is not always available from the user submission:**
   As discussed, in general, the user-submitted control-flow dependencies and the dataflow dependencies of a workflow do not have to be the same. Therefore, the dataflow information has to be gathered automatically during the computation.

2. **Traditional file systems do not track dataflow information:**
   The underlying file systems used in HPC typically do not track the dataflow information inherent to jobs. Historically, file systems react to file operations requested by the application instead of proactively gathering information.

To address these challenges, we propose a *WaFS* Scheduler (overview in Chapter 3.2; WaFS prototype discussed in Chapter 5), a novel approach that integrates the file systems and the batch schedulers to collect and exploit the dataflow information on a per-workflow instance (or per-instance, for short) basis. With this integration we can obtain several benefits:

1. The dataflow dependencies between the jobs in a workflow can be inferred by combining the scheduler's knowledge of the jobs (and possibly control-flow) and the file system's knowledge of the files accessed.

2. Separate namespaces can automatically be constructed on a per-instance basis to maximize the workflow instance concurrency while incurring low storage overhead, despite filename conflicts.

3. The dataflow information can be used to make trade-offs between concurrency and storage overhead when there are (potential) filename conflicts or deadlocks.

To achieve these ends, we propose and evaluate a set of dataflow-based scheduling policies, including *Versioned Namespace* (VNS) and *Overwrite-Safe Concurrency* (OSC), to address the problems of filename conflicts. A *hybrid* policy (HB) of combining VNS and OSC is also considered when storage resources are limited and deadlock is a potential problem. Using simulation studies for a variety of workloads, we show the value of dataflow-based scheduling policies for improving the degree of job concurrency and thereby decreasing makespan while minimizing the storage overhead of workflow-based computations. The HB policy for deadlock resolution is presented in the next chapter.

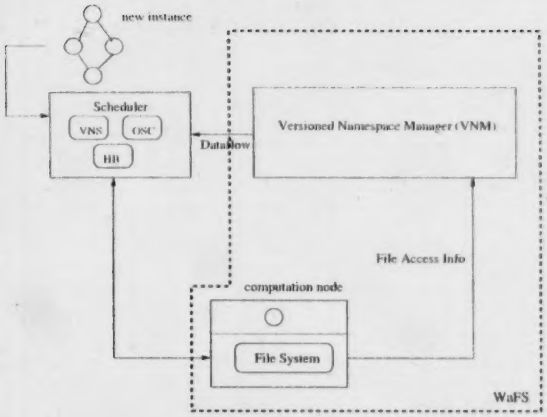


Figure 3.1. WaFS Scheduler: Integration of WaFS with Batch Scheduler for Dataflow Collection

3.2 Dataflow Collection: WaFS Scheduler

To collect the dataflow information and to manage a distinct namespace for each workflow instance, we propose a Workflow-aware File System that layers a *Versioned Namespace Manager* (VNM) on top of existing file systems and integrates it with the batch scheduler. The integrated system (i.e., WaFS + Batch Scheduler) is called *WaFS Scheduler*. Note that in traditional HPC systems neither the batch scheduler nor the file system can obtain and exploit the dataflow information alone. For example, file systems do not associate files being accessed with a workflow or instance; file systems passively respond to file operations without recording the jobs that access the files. Further, schedulers do not consider the set of files that a job, workflow or instance will access when making scheduling decisions.

The architecture of the WaFS Scheduler for dataflow collection is shown in Figure 3.1. It consists of two major components: the batch scheduler (enhanced with VNS, OSC or their hybrid HB policy) and WaFS. The enhanced batch scheduler obtains the dataflow information from WaFS and uses it to maximize job concurrency through the proposed policies (to be discussed in Chapter 3.3 and Chapter 4). WaFS monitors the workflow computations, and interacts with the underlying file system to capture the file access information and infer the dataflow information on a per-instance basis. More specifically, under the assumption that no filename conflicts occur inside workflow instances, for any pair of control-dependent jobs (i.e., where there is a direct path between two jobs in the control-flow graph), if a file is created by one job (source) and read by the other job (destination),

Policy	DOC	Intra-Instance limited by (Table 1.1)	Inter-Instance limited by (Table 1.1)	Storage Overhead	File Versioned	Storage Allocation/Deallocation Granularity
BASE	Low	Control flow	Control flow	Low	Never	Job/Instance
Sub-dir	High	Control flow	Total Storage	High	Always	Job/Instance
VNS	High	Dataflow	Total Storage	High	Always	Job/Job
OSC	Medium	Dataflow	Dataflow	Low	Never	Job/Job (when safe to overwrite)
HB	Selectable	Dataflow	Active Storage	Selectable	Selectable	Job/Job

Table 3.1. The Characteristics of the Compared Policies: VNS, OSC and HB are our dataflow-based policies, BASE is the control-flow-based serial policy, and Sub-dir refers to the policy that employs the working directory to address the filename conflicts and maximize the job concurrency. BASE and Sub-dir policies are listed for comparison purposes. DOC is short for "Degree of Concurrency."

then a data dependency is established between these two jobs (from the source job to the destination job). In addition, WaFS provides services for the batch schedulers to exploit the inferred dataflow information and maximize the job concurrency while minimizing storage overhead.

To validate the basic ideas behind WaFS, we developed a simple WaFS prototype (described in Chapter 5). The prototype works at the user level, using `ptrace()` via a monitor component (not shown in Figure 3.1; see Chapter 5), to trace the file-oriented system calls (e.g., `open()` and `close()`) and collect the dataflow information in the VNM. Although a full, production-quality dataflow-based scheduler has not been implemented, the WaFS prototype does validate the basic design and shows one possible implementation strategy of a key mechanism.

3.3 Dataflow-based Scheduling Policies

To exploit the WaFS mechanism, we propose three basic policies: Versioned Namespace (VNS), Overwrite-Safe Concurrency (OSC), and their hybrid (HB, detailed in Chapter 6). All three policies, together with the reference policies of *BASE* and *Sub-dir* are characterized in Table 3.1.

The essence of VNS and OSC is to exploit the dataflow information to selectively break the name dependencies (i.e., the filename conflicts) between concurrent workflow instances. Unlike HB, both policies assume that there are an arbitrary number of processors and storage resources. In other words, both policies study the limits to concurrency other than simple resource limits (e.g., data hazards [71]).

To simplify the presentation of VNS and OSC, we assume that the final output files are staged out

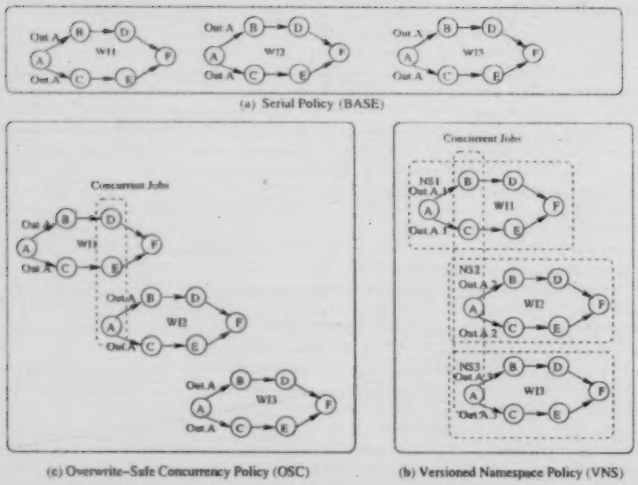


Figure 3.2. Inter-Workflow Instance Concurrency: (a) Serial Policy (BASE), (b) Versioned Namespace (VNS) and (c) Overwrite-Safe Concurrency (OSC)

to a different file system by the workflow instance itself before each instance is complete. Therefore, the WaFS Scheduler assumes it can simply deallocate *all* of the storage resources upon instance completion.

Consider Figures 3.2(b) and 3.2(c) as examples where three workflow instances (i.e., `WI1`, `WI2` and `WI3`) are submitted for scheduling. For comparison purposes, we also show the BASE policy i.e., the serial policy (Figure 3.2(a)). In BASE the inter-workflow instance concurrency is limited by the control-flow information and thus each workflow instance is executed sequentially (i.e., no inter-instance concurrency). Files are never versioned, and storage is deallocated after the completion of each instance (see Table 3.1). Although it is a bit of a "straw man" policy to execute the workflow instances sequentially, the BASE policy does represent a class of users and workloads in practice.

Perhaps a more reasonable comparison is the Sub-dir policy that employs a per-instance working directory strategy to isolate the input and output files of each individual workflow instance (i.e., files are essentially always versioned). Therefore, Sub-dir inherently breaks filename conflicts and increases the concurrency. In Sub-dir the inter-instance concurrency is limited by the available total storage, and the storage held by each instance is deallocated after the instance is completed (see Table 3.1). As we will see, Sub-dir is similar to VNS. In contrast, VNS is transparent to the application and also has other benefits discussed later.

3.3.1 Versioned Namespace (VNS) Policy

The VNS policy adopts a renaming strategy by automatically versioning each output file (Figure 3.2(b), VNS). Specifically, files are always versioned when created with a file open for writing. Then, when the file is closed and if the dataflow information determines that the file is no longer needed (e.g., has no more readers), the file storage is deallocated (see Table 3.1).

The basic strategy is similar to *register renaming* [48, 87] in processor microarchitecture in that extra (i.e., file) resources are used to improve concurrency. The major difference between VNS and register renaming is that the file-based dataflow information required for VNS to work is not readily available in current systems. Our proposed WaFS fills in that dataflow gap.

With VNS, although the different instances may generate files that have the same name, their version numbers are different. For example, in Figure 3.2(b), `Job A` in `WI1` and `WI2` may have output files that have the same name, `Out.A`, but this file will have different version numbers in each workflow instance, such as `Out.A.1` in `WI1` and `Out.A.2` in `WI2`. Given this versioning policy, together with the integration of the file system and job scheduler, VNS can construct a separate *namespace* for each workflow instance (i.e., `NS1`, `NS2` and `NS3`). Here, the namespace of VNS, in terms of isolating the workflow instances, is similar to the working directory in the Sub-dir policy (Chapter 1.1.1). In contrast, the namespace of VNS can be related back to the workflow instance (since scheduler and file systems are coupled) to capture and exploit its dataflow information.

First, with dataflow information, when a job is finished, VNS can delete the files that are no longer used immediately (i.e., job deallocation granularity; see Table 3.1). However, Sub-dir, without dataflow information, is unable to do so until the end of an instance, minimizing the effective storage utilization. Second, compared to the Sub-dir policy, VNS can potentially increase the *degree of concurrency* (DOC) (Chapter 2.3), since based on the dataflow information, VNS can improve the intra-instance concurrency by removing the implicit synchronizations (i.e., virtual job) in the control-flow DAG (see the implicit node in Figure 2.1 in Chapter 2).

3.3.2 Overwrite-Safe Concurrency (OSC) Policy

Even though, compared with the Sub-dir policy, the job deallocation granularity in VNS can reduce the storage overhead, the storage overhead of VNS is still high due to the potential of a large number of concurrent workflow instances.

To overcome the storage overhead of VNS, the OSC policy overwrites files when it is safe to do so instead of always versioning files as per VNS. Files are never versioned when created and never

deleted when closed, but they can be overwritten by later instances as long as they are not needed in the current instance (i.e., job deallocation granularity; see Table 3.1).

As an example, in Figure 3.2(c), Jobs `D` and `E` of `WI1` can execute concurrently with `Job A` of `WI2`. Specifically, `Job A` of `WI2` has to wait until the completion of both `Job B` and `Job C` of `WI1`, then `WI2`'s `Job A` can overwrite the file `Out.A`. The DOC increases to three (Figure 3.2(c): dashed box, concurrent jobs 'CJ2' limited only by dataflow information; see Table 3.1). Therefore, OSC improves the DOC as compared to the serial policy (Figure 3.2(a)) by increasing the inter-workflow instance concurrency.

Since OSC solves the filename conflict problem by overwriting files instead of versioning files in VNS, the storage overhead of OSC is small. In fact, the storage overhead of OSC is proportional to the actual DOC and not proportional to the number of workflow instances. On the other hand, OSC improves DOC over strategies (e.g., BASE) that must be conservative in overwriting files (e.g., when all jobs in a workflow instance are completed) but without incurring extra storage overhead.

3.3.3 Hybrid Policy (HB)

We note that VNS and OSC represent two extremes along a spectrum of policies that trade off storage overhead for DOC. Compared to VNS, OSC consumes much less storage, but its DOC is limited. Compared to OSC, VNS maximizes DOC, but it consumes much more storage. Ideally, a scheduler might want to use a combination of overwriting/deleting and versioning to selectively improve DOC while controlling storage overhead.

To achieve this goal, given a storage budget (i.e., the maximum amount of storage that can be used during the computation), our HB policy versions the output files for high DOC whenever deadlock (due to storage competition between multiple concurrent workflow instances) can be avoided. More specifically, when a job in instance i creates a file for writing, if there already exists a file with exactly the same name but no longer used in instance $i-1$, the existing file will be safely overwritten by instance i. Otherwise, a new version number for the created file will be obtained from VNM. In both cases, the new file is created without incurring deadlock. For example, in Figure 3.3, when `Out.A` is no longer used in `WI1`, it can be safely overwritten by `WI2`. However, a new version number for `Out.A` is needed (i.e., `Out.A.1`) for `WI3` since `Out.A` is being used in `WI2` when `WI3` starts.

Compared to VNS and OSC, HB can selectively control the storage overhead (e.g., via the storage budget) while increasing its DOC. In our example, it is not necessary for `WI3`, like in OSC, to wait for start until `Job B` and `Job C` in `WI2` are finished. Rather, if there is *sufficient* storage left

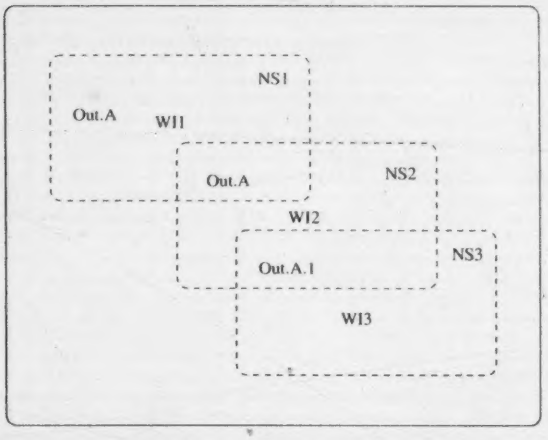


Figure 3.3. Inter-Workflow Instance Concurrency in HB Policy

(i.e., deadlock can be avoided), `WI3` can start immediately to improve DOC. Otherwise, `WI3` has to wait until sufficient storage is available, avoiding consuming too much storage as does in VNS. The comparison between the three proposed policies is summarized in Table 3.1 and the deadlock avoidance in the HB policy will be detailed in Chapter 4.

3.3.4 Summary

In this chapter, we proposed three basic policies, VNS and OSC and their hybrid (HB), to maximize job concurrency by addressing the problem of the control-flow-based batch schedulers (see Table 3.1).

VNS and Sub-dir are consistently the best overall policies in terms of DOC [99, 100], but both suffer from storage overhead. However, compared to the Sub-dir policy, VNS provides the benefit of being able to construct a namespace to infer and capture the dataflow information on a per-workflow instance basis. Therefore, VNS can improve the intra-instance concurrency and deallocate the unused storage at the earliest possible time. In contrast, without dataflow information Sub-dir can only deallocate the storage at the end of each instance, therefore suffering from more storage overhead than VNS.

Due to its low storage overhead, OSC is valuable in situations where storage is scarce, but it suffers from potentially lower DOC. HB is between VNS and OSC and can make more fine-grained trade-offs between DOC and storage overhead. Specifically, HB can optimize the performance by selective control of the DOC under a given storage budget.

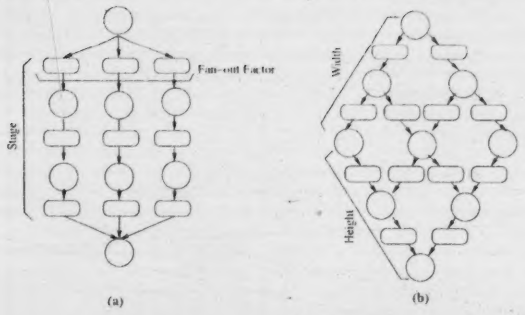


Figure 3.4. Benchmark Workflow Graphs: A circle represent a job, and a rounded rectangle represents an input/output file. The Fork&Join (a) is characterized by the fan-out factor and the number of stages, whereas the Lattice (b) is characterized by its height and width.

3.4 Simulation Results

We use simulation-based techniques to show the potential of dataflow information to improve workflow scheduling. In all experiments we use the serial policy and the Sub-dir policy, the two most common solutions in practice, as our baseline strategies (BASE and Sub-dir) and identify the circumstances under which OSC and VNS outperform these baseline strategies with respect to the *makespan*, the average DOC and the storage overhead.

VNS is equal to Sub-dir (current best practice) on makespan, but *always better*, and usually a factor of 2 or better, on storage overhead. OSC provides even more efficient storage utilization than either VNS or Sub-dir, while remaining comparable to VNS and Sub-dir on makespan for moderate to non-intensive workloads.

With WaFS, depending on the workloads being studied, OSC and VNS can substantially improve makespan over BASE usually by an order-of-magnitude. The actual improvement depends on the arrival rate of the workload and other factors. To different degrees, OSC and VNS exploit the inherent concurrency between workflow instances that BASE is unable to exploit.

3.4.1 Methodology

Based on the applications described in Chapter 2, we chose to use three representative structures: Fork&Join, Lattice and Pipeline (see Figure 3.4). These structures cover a spectrum of workflows and DOC. The Fork&Join structure, characterized by the number of stages and fan-out factors, exhibits near-constant DOC and is representative of a large class of problems with a Pipeline of parallel

phases [15, 28, 91, 98]. The Lattice structure, characterized by its width and height, exhibits variable concurrency, where the concurrency increases initially to a maximum degree and then decreases progressively. A variety of numerical linear algebra computations that arise in a broad range of scientific and engineering applications have a Lattice structure [38, 59, 78, 81]. The Pipeline structure can be viewed as a special case of Fork&Join (i.e., fan-out factor is one) or Lattice (i.e., either width or height is one), but it is very common in scientific computation [6, 20, 42, 51, 89].

Example dataflow DAGs for these workflows are shown in Figure 3.4. For the Fork&Join, we assume that the control-flow DAG is similar to its dataflow counterpart except that any two consecutive stages (i.e., all jobs in the stage) are synchronized by an implicit virtual job (see Figure 2.1). In contrast, for both Lattice and Pipeline, the control-flow DAG and the dataflow DAG are assumed to be exactly the same. Although user-submitted control-flow DAGs may have various shapes, the assumptions we made here are reasonable for users to easily reason about their workflows.

As implied in the previous sections, for the OSC and VNS strategies to work, the scheduler must know both the control-flow of the computation (i.e., the control-flow DAG) and the dataflow of the jobs in the workflow. Control-flow information is the typical way in which dependencies are made known to batch schedulers such as LSF [108], PBS [47] and Condor [18]. Dataflow information is gathered by the WaFS during the execution of the first workflow instance and exploited by OSC and VNS to improve the DOC of later instances.

Since there are no well-accepted models for job service times (JST), data file sizes (FS) nor their relationships for the workflow-based workloads, in experiments we assume that for instances of all the examined workflows, the job service time as well as the data file size are uniformly distributed. These assumptions are consistent with some previous studies [16, 88, 106, 107]. A brief examination of the non-uniform Zipf distribution for JST (and file sizes) can be found in Appendix B. In addition, in each experiment, there are a total of 100 workflow instances in the workload, and the workflow instance inter-arrival time follows the exponential distribution.

The characteristics of the benchmark workloads are summarized in Table 3.2, and the compared policies, except for the HB policy, are characterized in Table 3.1. We do not evaluate the HB policy since it is identical to the VNS policy when the storage budget is not a concern. We study the HB policy with a variety of deadlock avoidance algorithms in Chapter 4.

We further assume that an unbounded number of homogeneous computational nodes and infinite storage are available so that the maximum DOC is never constrained by the hardware.

We use the discrete event simulation package SMURPH [33] to implement our simulator. The simulated scheduler is given the control-flow DAG by the user submitting the workflow instances.

Characteristics	Fork&Join	Lattice	Pipeline
Shape Parameter	stages × fan-out	height× width	stages
Job Service Time	uniform	uniform	uniform
Inter-arrival Time	exponential	exponential	exponential
Workload Size	100	100	100
File Size	uniform	uniform	uniform

Table 3.2. The Characteristics of the Benchmark Workloads

A simulated Versioned Namespace Manager (i.e., VNM) sees all of the file reads and writes and records the dataflow DAG for a workflow. Based on the historical dataflow information, the scheduler knows (from VNM) the dataflow of each workflow instance. The SMURPH-based simulation is written in C++ with both the versioned namespace manager and scheduler abstracted into modules independent of the underlying simulation engine.

3.4.2 Results, Data Points and Standard Deviation

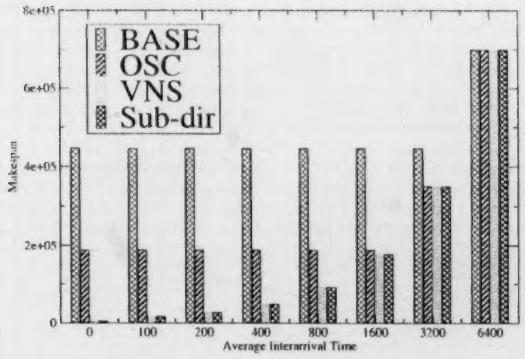
There are a variety of factors that impact the performance and average DOC of the workloads (i.e., makespan and average DOC). Some of those are identified in our experiments as follows:

1. *Instance Inter-arrival Time Distribution:* simulated user behavior, (e.g., exponential distribution).
2. *Workflow Shape:* the structure of the workflow (e.g., Pipeline).
3. *Job Service Time* (JST), simulated job behavior (e.g., uniform distribution).

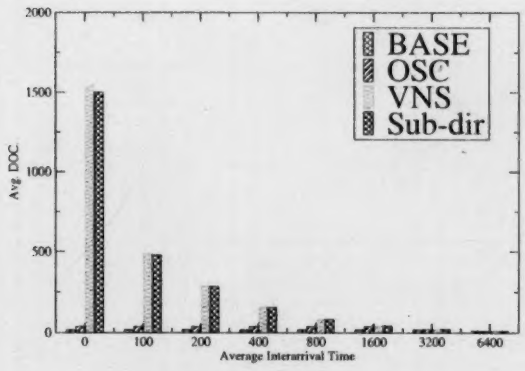
Since the storage budget (assumed in this chapter) is unbounded, the file size distribution does not affect the makespan and the average DOC; it only affects the storage overhead. Therefore, in all experiments we fix the file size distribution as a uniform distribution on [1,10] storage units. The data point in each experiment is averaged over 10 runs by changing the random seed in the simulator.

We found that in all experiments reported in this chapter, the standard deviation for the 10 runs is never greater than 12% of the mean of the 10 runs (i.e., the data point's value). More specifically, for all makespan and DOC data points, the standard deviation is less than 5%, and for all storage overhead data points, the standard deviation is less than 12%. Therefore, for clarity of presentation, we do not show the standard deviation bars on the graphs. A similar presentation strategy is discussed in,Chapter 4.5.4.

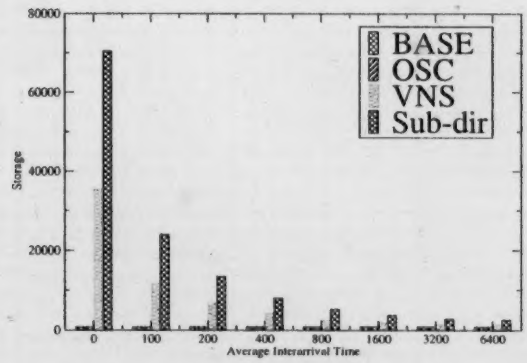
We first vary the average inter-arrival time of workflow instances to understand their impact on performance and storage overhead. For a Fork&Join structure with three stages and a fan-out of



(a) Makespan



(b) Average DOC.



(c) Peak Storage Overhead

Figure 3.5. Simulation Results for the Fork&Join (3×32): (a) Makespan, (b) Average DOC and (c) Storage Overhead. (DOC units are numbers of jobs; all other values are either time units or storage units)

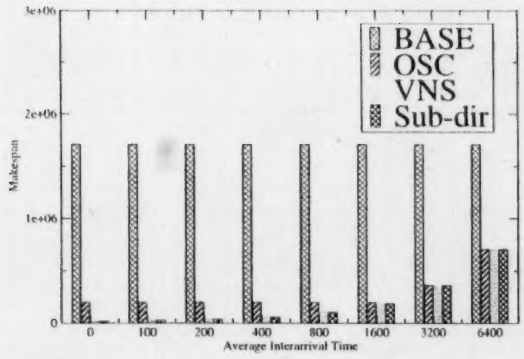
32 per stage. Figure 3.5 shows makespans, corresponding average DOC and storage overhead for a variety of different simulation parameters. In Figure 3.5, the JSTs are uniformly distributed between 500 and 1000 time units, and we vary the inter-arrival time between instances from 0 to 6400 time units. Intuitively, a short inter-arrival time represents an intense workload, where the instances arrive close to each other. On the extreme right of each graph, an inter-arrival time of 6400 represents a lighter workload, where the inter-arrival time is much larger than the job service times.

For intensive workloads (i.e., x-axis $\leq$ 200 in Figure 3.5(a)), VNS and Sub-dir are better (i.e., lower makespan) than BASE (i.e., the typical, Serial Strategy) by over an order-of-magnitude. OSC also has a lower makespan than BASE, but not as low as VNS. The performance improvements are due to improvements in the DOC (Figure 3.5(b)), which typically results in a lower makespan. As discussed earlier, VNS isolates each workflow instance by creating a separate namespace for each instance. As a result, there are no name conflicts between the different instances, and the jobs can be executed immediately as long as their intra-workflow instance data dependencies are respected. Sub-dir creates a separate directory for each instance and thus has similar performance to VNS. However, compared with VNS, Sub-dir has a somewhat lower DOC due to its control-flow based scheduling (VNS is based on the dataflow), especially when all the instances in a workload arrive at the same time (i.e., x-axis 0). However, this difference is marginal.

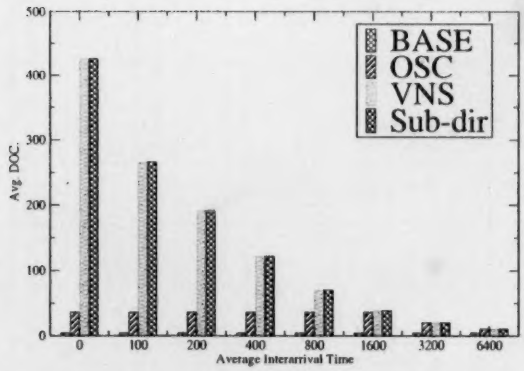
The main drawback of Sub-dir is its storage overhead since it never overwrites files until the end of the instance computation. In contrast, both BASE and OSC create only a limited number of different files for the workload, and VNS can overwrite files immediately based on the data dependency information.

As the instance inter-arrival time increases (i.e., the arrival rate decreases), the performance difference as well as the storage overhead between BASE, OSC, VNS and Sub-dir diminish. A larger inter-arrival time means that fewer workflow instances are in the scheduler's queue at any given time, which implies a smaller number of active instances and a smaller DOC. Since the storage overhead of the compared policies is either proportional to the number of active instances (Sub-dir and VNS) or proportional to DOC (i.e., BASE and OSC), it decreases as the instance inter-arrival time increases. Naturally, if there is a lack of inherent job concurrency in the workload, the benefits of OSC and VNS are not observed.

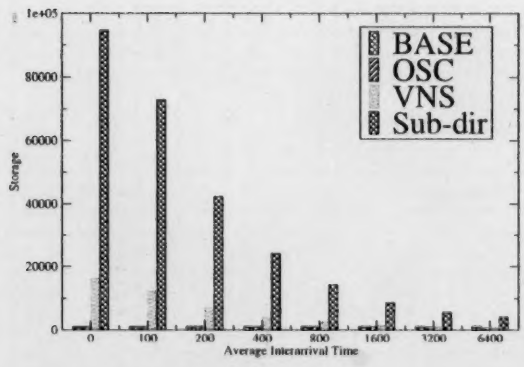
Therefore, for low-arrival-rate workloads (e.g., where the inter-arrival time is 3200 time units or larger), the BASE strategy is preferred since it has the same makespan of the other strategies, with none of the additional complexity and overhead. For medium-arrival-rate workloads (e.g., where the inter-arrival time is between 1200 and 2400 time units), OSC performs almost as well as VNS



(a) Makespan



(b) Average DOC.



(c) Peak Storage Overhead

Figure 3.6. Simulation Results for the Lattice (8×12): (a) Makespan, (b) Average DOC and (c) Storage Overhead. (DOC units are numbers of jobs; all other values are either time units or storage units)

and Sub-dir, but without the storage overhead. For high-arrival-rate workloads (e.g., where the inter-arrival time is 800 time units or less), VNS is the clear performance leader. It outperforms Sub-dir in terms of makespan and storage overhead.

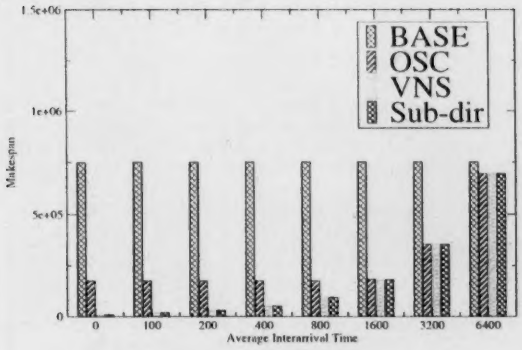
Many HPC workloads consist of a large parameter sweep, where all workflow instances are known at the beginning of the computation, corresponding to inter-arrival times of 200 time units (or less). This also corresponds to the region of the graphs where OSC and VNS perform best.

To evaluate the impacts of workflow shapes, we did the same simulation studies on both the Lattice and Pipeline workflow whose simulation results are shown in Figures 3.6 and 3.7, respectively. Recall that the Lattice is expected to have a lower intra-workflow instance DOC than the Fork&Join because of the additional dependencies between the jobs. For our specific Lattice, an 8 $\times$ 12 rectangle/diamond, the critical path through each workflow instance is much longer than the 3-stage Fork&Join discussed above. This is reflected in the near-constant makespan for BASE despite variations in the inter-arrival times of the workflow instances. Intuitively, the Lattice has a lower average DOC than the 3-stage Fork&Join and a longer critical path, which reduces the intra-workflow instance DOC such that the BASE strategy cannot reduce the makespan, even for low-arrival-rate workloads.

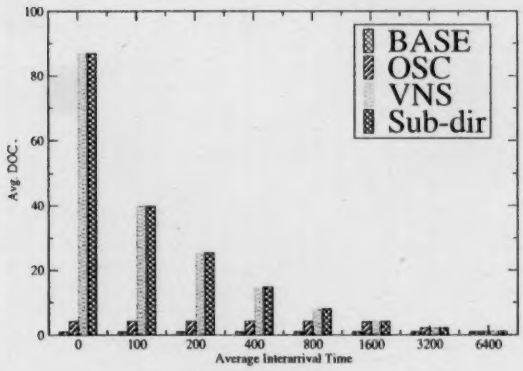
However, both OSC and VNS can still exploit inter-workflow instance concurrency to significantly reduce makespans through higher DOC. VNS continues to be better than OSC at reducing the makespan, but (once again) at the cost of increased file storage due to versioning. Sub-dir demonstrates the same performance as VNS since the shapes of the control-flow DAG and the dataflow DAG are exactly the same for our Lattice workflow. However, Sub-dir suffers from larger storage overhead than VNS.

For both VNS and Sub-dir, their performance improvements over BASE are largely independent of the workflow shapes. This is different from OSC. For OSC a longer critical path usually implies a larger number of concurrent instances during the computation. So OSC exhibits relatively better performance for a workflow with a longer critical path. We can observe this by comparing the makespans between BASE and OSC in Figures 3.5(a) and 3.6(a), where, again, the critical path of the Lattice instance is much longer than that of the Fork&Join instance.

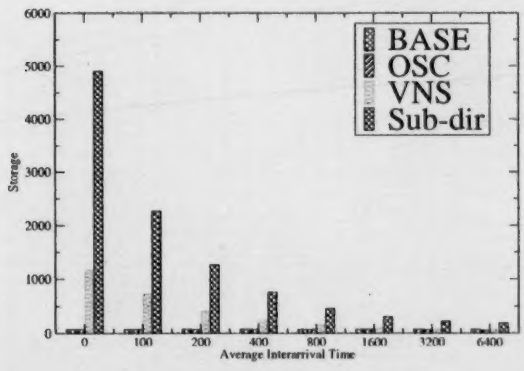
In contrast to the 3-stage Fork&Join, we also found that the difference of storage overhead between VNS and Sub-dir for Lattice becomes relatively large (compare Figures 3.5(c) and 3.6(c)). This is not difficult to understand since DOC is proportional to the storage overhead of VNS, and the DOC of the Lattice is much less than that of the Fork&Join (i.e., the storage overhead of VNS for Fork&Join is relatively high).



(a) Makespan



(b) Average DOC.



(c) Peak Storage Overhead

Figure 3.7. Simulation Results for the Pipeline (10-stage): (a) Makespan, (b) Average DOC and (c) Storage Overhead. (DOC units are numbers of jobs; all other values are either time units or storage units)

The same performance observation concerning the compared policies can also be observed in the graphs for the 10-stage Pipeline workflow (see Figure 3.7), an extreme case of Fork&Join (Lattice) workflow. However, the relative performance between OSC and BASE for the Pipeline is not as good as that for the Lattice (compare Figures 3.6(a) and 3.7(a)). This is easy to understand since there is no intra-workflow instance DOC, the significant performance improvements of OSC, VNS and Sub-dir over BASE are derived totally from exploiting the inter-workflow instance job concurrency. On the other hand, in our experiments the critical paths of the Pipeline instances are shorter than those of the Lattice instances, limiting the number of concurrent instances for OSC.

Since the intra-instance concurrency of the Pipeline workflow is lower, the difference in storage overhead between VNS and Sub-dir is relatively large for the Pipeline, which is similar to the situation with the Lattice.

To summarize, for all the benchmark workflow shapes we have the following conclusions:

1. Depending on the workloads being studied, OSC and VNS consistently outperform BASE by up to an order-of-magnitude. Most performance gains are from exploiting inter-instance concurrency.

2. VNS continues to be better than OSC at reducing makespan but at the expense of increased file storage. Sub-dir has *almost* the same performance as VNS but suffers from larger storage overhead than VNS. The performance improvements of both policies over BASE are independent of the workflow shapes.

3. The workflow shape impacts the performance of OSC. In general, OSC exhibits better performance for a workflow with a longer critical path.

4. VNS in general is more efficient than Sub-dir in terms of storage utilization. The relative difference in storage overhead between VNS and Sub-dir depends on the amount of intra-instance concurrency. When intra-instance concurrency is highest, the difference is lowest.

In the following experiments (Figures 3.8, 3.9 and 3.10), we show how DOC, makespan and storage overhead depend on multiple factors, including the job service time (JST), the shape of the workflow DAG and the instance inter-arrival time. We tried various JST ranges to approximate poorly-balanced job service time (i.e., JST in the range of [10, 1000]), moderately-balanced job service time (i.e., JST in the range of [500, 1000]; see the previous experiments), and well-balanced job service time (i.e., JST in the range of [800, 1000]). We also varied the inter-arrival time between 0 and 6400 time units.

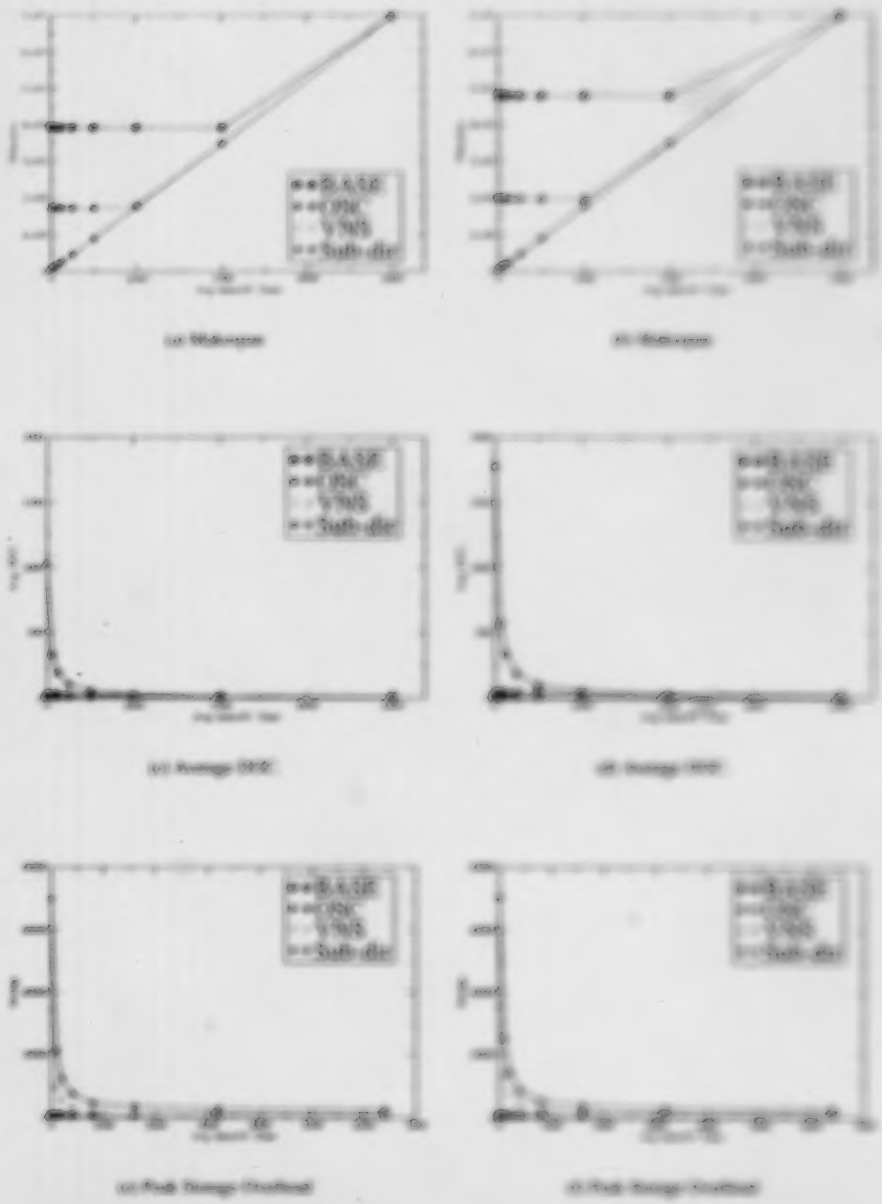


Figure 5.8. Impacts of Job Service Time on the ForkSLAVA (0 - 10): Makespan, Average DOC and Storage Overhead (Left: JST(10, 1000), Right: JST(500, 1000))

Our conclusions from these experiments are:

1. Independent of JST and the shape of the workflow DAG, OSC and VNS are consistently better than BASE and Sub-dir for high-arrival-rate workloads (i.e., low inter-arrival times) with respect to makespan and storage overhead, respectively.

2. As the JST range varies, the inherent intra-instance DOC of the workload changes (as expected, except for the Pipeline) because processes are left idle due to the load imbalance. However, OSC, VNS and Sub-dir continue to achieve higher DOC than that achieved by BASE, for intensive workloads, at the expense of increased storage overhead.

3. Ultimately, the maximum JST (which is always 1000 time units) in our experiments determines the critical path of each workflow instance and thus the makespan. Consequently, regardless of the load imbalance *within* workflow instances, OSC, VNS and Sub-dir exploit enough concurrency *between* workflow instances to be preferred over BASE, with similar caveats and trade-offs as discussed for Figure 3.5.

4. The impact of JST on the storage overhead of each compared policy is different. Specifically, for intensive workloads, regardless of the workflow shape, the impact on BASE, OSC, and VNS is small. However, depending on the workflow shape, the impact on Sub-dir is different, either small for both the Fork&Join and Lattice or large for the Pipeline (compare Figures 3.10(e) and 3.10(f)).

3.4.3 Summary

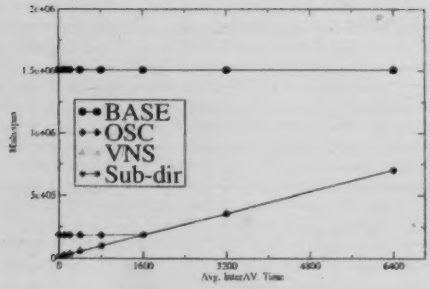
Our simulation studies show that the basic idea of the WaFS Scheduler (i.e., the integrated file system and batch scheduler) can effectively resolve filename conflicts and significantly improve job scheduling by maximizing job concurrency while lowering storage overhead. Specifically, gathering and using dataflow information to support the novel OSC and VNS scheduling policies is shown to:

1. reduce makespans, relative to BASE;

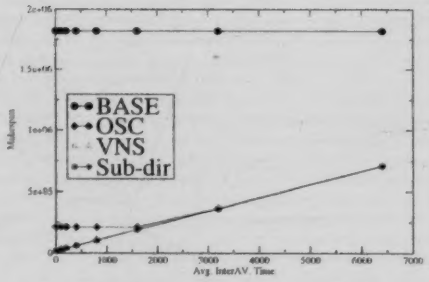
2. reduce storage overhead, relative to Sub-dir;

3. improve inter-workflow instance concurrency, relative to BASE;

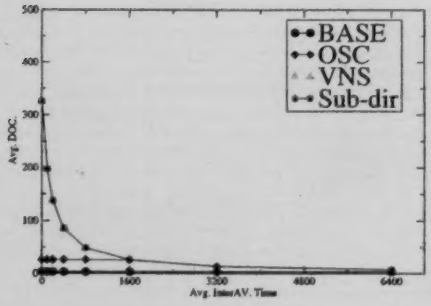
4. maintain the benefits over BASE and Sub-dir for a variety of workload intensities, a variety of job service time distributions and three different typical workflow shapes (i.e., Fork&Join, Lattice and Pipeline).



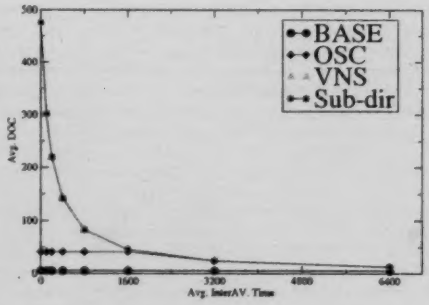
(a) Makespan



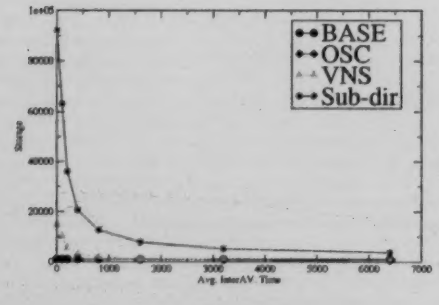
(b) Makespan



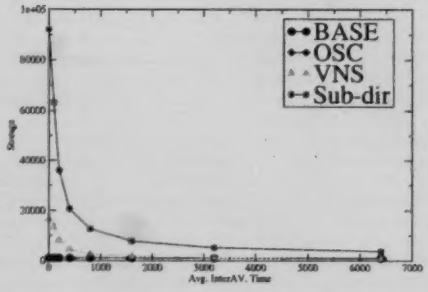
(c) Average DOC.



(d) Average DOC.

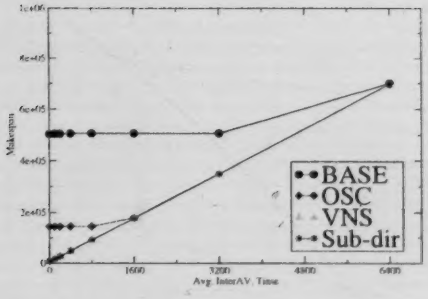


(e) Peak Storage Overhead

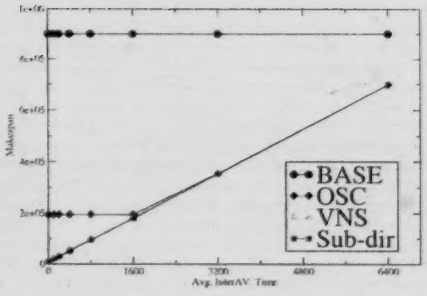


(f) Peak Storage Overhead

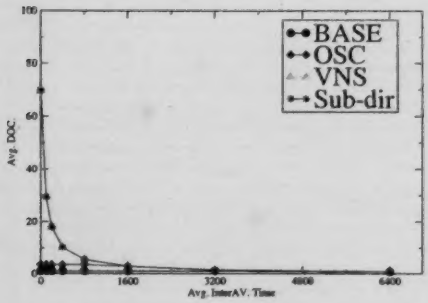
Figure 3.9. Impacts of Job Service Time on the Lattice (8×12): Makespan, Average DOC and Storage Overhead (Left: JST[10, 1000], Right: JST[800, 1000]).



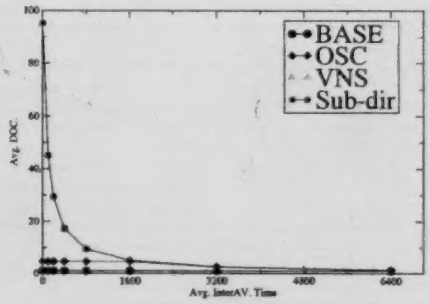
(a) Makespan



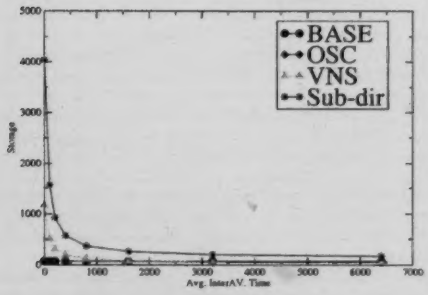
(b) Makespan



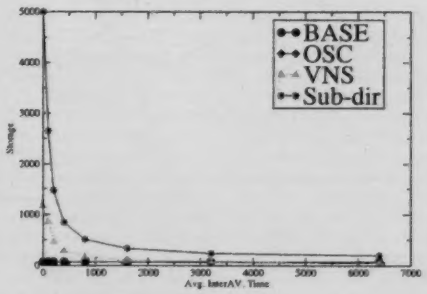
(c) Average DOC.



(d) Average DOC.



(e) Peak Storage Overhead



(f) Peak Storage Overhead

Figure 3.10. Impacts of Job Service Time on the Pipeline (10-stage): Makespan, Average DOC and Storage Overhead (Left: JST[10, 1000], Right: JST[800, 1000]).

The main criteria for choosing between OSC and VNS is the trade-offs in performance for the storage overhead of file versioning.

3.5 Concluding Remarks

In this chapter we studied the potential of using dataflow information in maximizing job concurrency while resolving filename conflicts. Our contributions are:

1. We propose the WaFS Scheduler, a novel approach that integrates the file systems with the batch schedulers to collect the dataflow information and make it available to the control-flow-driven batch scheduler in order to facilitate the workflow scheduling.

2. To exploit the inferred dataflow information, we propose and evaluate through simulation studies a set of simple yet effective scheduling policies: VNS, OSC and their hybrid (HB) (details in Chapter 4). The essence of these policies is to take advantage of the dataflow information to remove artificial limits (i.e., filename conflicts) on the degree of concurrency and thereby to allow the batch scheduler to better exploit the available HPC resources.

Our simulation results show that by combining dataflow information with a versioned namespace (i.e., VNS), depending on the workload, the makespans can be improved by over an order-of-magnitude, while the storage overhead is low. In addition, the dataflow information can also make trade-offs between concurrency and storage overhead (i.e., OSC) when there are (potential) filename conflicts.

Chapter 4

Dataflow-based Scheduling for Deadlock Avoidance

In this chapter we study the *hybrid* (HB) policy that combines versioning and overwriting, and leverages the dataflow information to address the deadlock problem when storage resources are constrained. To this end, we integrate two novel concepts with the traditional problem of deadlock avoidance. First, we show how knowledge of dataflow information can be exploited at runtime to improve the banker's-based algorithms and also reduce makespan. Second, we show how a distinction between *active* and *inactive* resources, rather than allocated versus un-allocated resources, is important to minimizing makespan.

Although the density and availability of storage is increasing rapidly, most HPC centers still operate with disk quotas in some form or another. In practice, storage is still a finite resource [76]. In fact, given the ease with which parameter-based studies can generate jobs and workflow instances, storage can often be a constraining resource. Therefore, a scheduler should both deal with potential deadlock issues and maximize performance as measured by *makespan* and *throughput*.

As discussed in Chapter 2, deadlock can be handled by prevention, avoidance, or detection. In comparison to prevention, deadlock avoidance has more potential to make effective use of storage since decisions are made dynamically. And in comparison to detection, deadlock avoidance does not have the deadlock recovery overhead. However, deadlock avoidance requires knowledge of the storage requirements of the computations, which can either be given by the user (e.g., as part of the job description) or estimated by the system based on historical information.

We propose two algorithms for deadlock avoidance that attempt to maximize active (not just al-

located) resource utilization and minimize makespan. Our approach is based on the well-known banker's algorithm, but our algorithms make the important distinction between active and inactive resources, which is not a part of previous approaches. The central idea is to leverage the gathered dataflow information to dynamically approximate *localized maximum claim* (i.e., the resource requirements of the remaining jobs of the instance) to improve either inter-instance or intra-instance concurrency and still avoid deadlock.

There are two primary, new algorithms. First, with *Dataflow-based Aggregate Requests* (DAR), the maximum claim of each job is dynamically computed by summing the resource requirements of all the remaining jobs (i.e., those jobs that have not yet been finished), instead of using a static predefined value. Second, the *Dataflow-based Topological Ordering* (DTO) algorithm exploits dataflow knowledge to topologically order (i.e., a specific order of job completion that is within a resource budget) the jobs when checking for safety. For both algorithms, the computed localized maximum claims are either independent of the scheduling orders of the remaining jobs (DAR) or not (DTO), with different advantages and disadvantages.

In simulation-based studies we integrate both algorithms into a dataflow-driven batch scheduler and show how DAR and DTO are better than the banker's algorithm and Lang's algorithm [60] in terms of makespan and active storage utilization. A variety of workflow shapes and parameters are examined. Depending on the situations, either DAR or DTO was found to be the best algorithm.

In addition, we also investigate the behavior of the proposed algorithms and show how dataflow information can be used to integrate an instance admission control with the deadlock avoidance algorithms to further reduce the makespan.

4.1 Notation and Workflow Model

To describe the proposed algorithms, we first summarize some notation in Table 4.1 and then define our *workflow model*. In our workflow model, the dataflow graph of a workflow as presented in Chapter 2 is refined as a weighted DAG $G(N, E)$, where N is a set of n nodes and E is a set of e edges. A node in the DAG represents a job which in turn is a program that must be executed in sequential order. The weight of the node is the job's service time. The weight of the edge indicates the size of the file that is created by the source node and used by the destination node. The precedence constraints of a DAG dictate that a node cannot begin execution until all its input files have arrived and no output files are available until the node has finished, and at that time, all output files are simultaneously accessible to its destination nodes. Hereafter, we use the terms node and job

symbol	*meaning*
m	the number of instances in a workload
n	the number of jobs in instance; it is a constant from instance to instance.
$r(t)$	the available storage (i.e., the free storage) at moment t
J_j^i	the jth job in J^i
J^i	the set of the jobs in instance I_i (i.e., $\{J_0^i, ..., J_{n-1}^i\}$)
$\|W_j^i\|$	the size of the write data set of job J_j in instance I_i
$\|R_j^i\|$	the size of the read data set of job J_j in instance I_i
$S_c^i(t)$	the set of the completed jobs in instance I_i before moment t
$S_r^i(t)$	the set of the ready jobs in instance I_i at moment t
$S_a^i(t)$	the set of the active jobs in instance I_i at moment t
$S_s^i(t)$	the set of the scheduled jobs in instance I_i at moment t, i.e., $S_s^i(t) = S_c^i(t) \cup S_a^i(t)$

Table 4.1. Notation Used in Algorithm Descriptions

interchangeably for easy presentation.

A workload may consist of multiple instances of the same workflow, with each instance having its own node and edge weights. Thus, for an instance $I_i, 1 \leq i \leq m$ (i is the index of I_i and m is the number of instances in a workload), the read and write data sets of the job J_j^i can be denoted as $R_j^i = \{r_{j1}^i, r_{j2}^i, ..., r_{jk}^i\}$ and $W_j^i = \{w_{j1}^i, w_{j2}^i, ..., w_{jl}^i\}, 1 \leq j \leq n$ (where n is the number of jobs in the instance, r and w represent input files and output files, respectively. l and k are integers). The total sizes of R_j^i and W_j^i, denoted by $|R_j^i|$ and $|W_j^i|$, respectively, are defined as $|R_j^i| = \sum_{s=1}^{k} |r_{js}^i|$ and $|W_j^i| = \sum_{s=1}^{l} |w_{js}^i|, 1 \leq j \leq n$, respectively; where $|f|$ represents the size of a file f.

In practice, the node and edge weights are generally estimated by the user, but the actual values may be different than the estimated values. For deadlock avoidance consideration, we tend to be conservative here by assuming that both the node and edge weights are always over-estimated. The claim on the storage for a job J_j^i in instance I_i is, therefore, known *a priori* to the scheduler and can be computed as $|R_j^i| + |W_j^i|, 1 \leq j \leq n$. However, due to data dependencies, the claim of job J_j^i can be simplified as $|W_j^i|, 1 \leq j \leq n$ since the input storage has been allocated by its parent jobs. Similarly, we can define the storage that job J_j^i can release after it has completed as $|R_j^i|, 1 \leq j \leq n$, if each file is safely deleted based on dataflow information (e.g., we may use a reference counter to record if a file is read by multiple jobs, and only those files whose reference counters are zero are in R_j^i).

Without loss of generality, a single source node (i.e., top node) and a single sink node (i.e., bottom node) are assumed in the DAG. These two nodes can be viewed as the jobs in the workflow that stage in the initial input files and stage out the final output files, respectively.

During the execution of a workflow instance, the life cycle of a job may experience several states, as discussed in Chapter 2. A completed job will release the storage space of its input files only if

Algorithm	Graph Type	Maximum Claim Scope, Computation Time	Safe Sequence	Storage Allocation/ Deallocation Unit
Banker's	Control-flow	Global, Static	Instance	Instance
DAR	Dataflow	Local, Dynamic	Instance	Instance
DTO	Dataflow	Local, Dynamic	Jobs of the current instance	Job

Table 4.2. The Characteristics of the Compared Deadlock Avoidance Algorithms

they are no longer used by other jobs (i.e., under the dataflow-driven batch scheduler). The released storage space can be reallocated to other jobs or instances. The output files are not released; they are kept as the input files to later jobs.

Our model is deterministic, at least to the extent that the time, and storage space required by any job, as well as the data dependencies among the jobs, are pre-determined and remain unchanged during the computation.

4.2 The Algorithms

As stated, both our proposed algorithms, DAR and DTO, are based on the banker's algorithm. However, previous deadlock avoidance algorithms do not distinguish between active versus inactive resource utilization. In contrast, DAR and DTO attempt to improve (makespan) performance by maximizing active resource utilization.

Table 4.2 summarizes the major characteristics of DAR and DTO. For comparison purposes, a control-flow-based banker's algorithm is also included. The major difference between our algorithms and the banker's algorithm is that our algorithms are based on dataflow instead of control-flow. With DAR and DTO it is possible to locally and dynamically compute maximum (resource) claims since it is possible to determine when resources are deallocated. Therefore, for subgraphs of the dataflow graph the maximum claims will be lower (technically, monotonically non-increasing, possibly decreasing) than for the entire graph. But, since the control-flow graph does not record when resources are deallocated, all subgraphs necessarily have the same maximum claim values, which are aggregates of *all* job requests. DAR and DTO are different from each other in the scope (i.e., jobs within the current instance versus all instances in the workload) they use to construct the safe sequence (jobs or instances) for a safety check. More details of both algorithms are presented in Chapter 4.2.1 and Chapter 4.2.2, respectively. The notation used in the descriptions are summarized in Table 4.1.

4.2.1 The DAR Algorithm

The motivation of the DAR algorithm is to try to maximize active storage utilization. Specifically, DAR tries to improve inter-instance concurrency. To this end, we view each workflow instance in the workload as a unit of storage allocation and deallocation (in contrast to DTO, Chapter 4.2.2) and leverage the dataflow information to minimize the maximum claim associated with each instance at runtime. By dynamically reducing the maximum claim associated with each instance, the goal is to increase the number of instances running concurrently. More specifically, in the DAR algorithm we have previously defined the maximum claim of an instance I_i (i is the index of instance I_i) at moment t to be the total requests of all the remaining jobs (i.e., those jobs in $J^i - S_c^i(t)$, which contains the jobs that have not finished. Here, J^i represents the total jobs in I_i and $S_c^i(t)$ represents the set of completed jobs in I_i before moment t, see Table 4.1.) in the instance I_i.

Formally, we can define it as

$$max\_claim(i,t) = \sum_{J_j^i \in (J^i - S_c^i(t))} |R_j^i| \tag{4.1}$$

Initially (i.e., $t = 0$), the set of completed jobs, $S_c^i(0) = \phi$, the maximum claim associated with the instance I_i represents the total requests of *all* the jobs (i.e., the banker's algorithm's maximum claim). More formally, based on our workflow model, we have $max\_claim(i,0) = \sum_{J_j^i \in J^i} |R_j^i| = \sum_{J_j^i \in J^i} |W_j^i|$ if only distinct files in instance I_i are counted (i.e., if a file is read by multiple jobs, the size of the file is only counted once in $max\_claim(i,t)$).

In this way, the maximum claim is localized and monotonically non-increasing (possibly decreasing) as the execution of the instance proceeds (i.e., dynamic computation, where $max\_claim(i,t)$ changes for different t). This is different from the standard control-flow-based banker's algorithm where a global maximum claim is pre-computed by conservatively aggregating all the jobs' requests. (i.e., $max\_claim(i,t)$ is a constant for all t)

Figure 4.1 is an example showing how the maximum claims defined in DAR are computed. In the example, three workflow instances with different shapes, Pipeline (Figure 4.1 (a)), Fork&Join (Figure 4.1 (b)) and Lattice (Figure 4.1 (c)), are considered. The jobs (represented by the nodes) inside the dashed regions have not been completed as of time t'. They are either active or unscheduled.

By applying Equation (4.1), the computation of the maximum claims for the three workflow instances at $t = 0$ and $t = t'$ can be done. These are detailed in Table 4.3. Note that some jobs have been completed as of time t' (i.e., the jobs outside the dashed regions in Figure 4.1). We can see

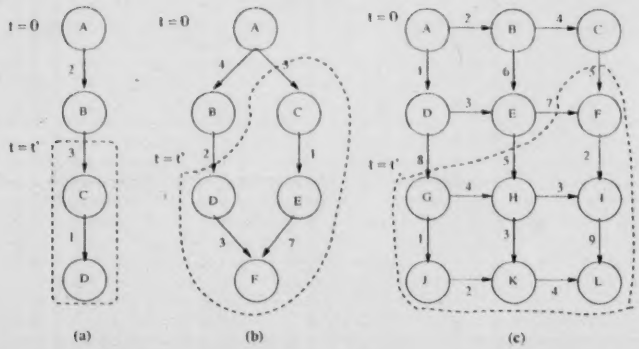


Figure 4.1. An example showing how the maximum claims defined in DAR are computed. In this example three workflow instances I_is with different shapes, (a) Pipeline, (b) Fork&Join and (c) Lattice, are considered. The nodes in the graphs represent the jobs, and the jobs inside the dashed regions are those jobs that have not been completed as of time t'. The numbers marked beside each edge indicate the file sizes.

that the maximum claims are monotonically decreasing in this example (non-increasing in general) as the instances proceed, regardless of the workflow shapes.

Based on the definition in Equation (4.1), we have the following lemma:

Lemma 4.2.1 *Given a workflow instance I_i where each file is only read by one job and no files created by some completed jobs will be read by a subset of unscheduled jobs in $J^i - (S_s^i(t) \cup S_r^i(t))$ at moment t, the need matrix, defined as the difference between the maximum claim and the allocated resources, can be computed as*

$$need(i,t) = \sum_{J_j^i \in J^i - S_a^i(t)} |W_j^i| \tag{4.2}$$

where, at the moment t, the set of completed jobs is $S_c^i(t)$, the set of active jobs is $S_a^i(t)$ and the set of ready jobs is $S_r^i(t)$.

Proof The allocated storage for instance I_i at moment t is computed as

$$alloc(i,t) = \sum_{J_j^i \in (S_a^i(t) \cup S_r^i(t))} |R_j^i| + \sum_{J_j^i \in S_a^i(t)} |W_j^i| \tag{4.3}$$

Since the intersection of any two sets in $\{S_c^i(t), S_a^i(t), S_r^i(t)\}$ is empty, we have $J^i - S_c^i(t) =$

Workflow Instance with Different Shapes	Time (t)	Completed Jobs ($S_c^i(t)$)	Uncompleted Jobs ($J^i - S_c^i(t)$)	$max\_claim(i,t)$
Pipeline I_i	0	ϕ	$\{A,B,C,D\}$	$2+3+1=6$
	t'	$\{A,B\}$	$\{C,D\}$	$3+1=4$
Fork&Join I_i	0	ϕ	$\{A,B,C,D,E,F\}$	$4+3+2+1+3+7=20$
	t'	$\{A,B\}$	$\{C,D,E,F\}$	$3+2+1+3+7=16$
Lattice I_i	0	ϕ	$\{A,B,C,D,E,F,$ $G,H,I,J,K,L\}$	$2+4+1+6+...+4=$ 69 (all edge weights)
	t'	$\{A,B,C,C,E\}$	$\{F,G,H,I,J,K,L\}$	$5+7+5+8+2+4+3+$ $1+3+9+2+4=53$

Table 4.3. The computation of the maximum claims are detailed for the three examined workflow instances at $t = 0, t'$ where, as of time t', some jobs have been completed in each instance.

$S_a^i(t) \cup S_r^i(t) \cup (J - S_c^i(t) - (S_a^i(t) \cup S_r^i(t)))$, and

$$
\begin{aligned}
need(i,t) &= max\_claim(i,t) - alloc(i,t) \\
&= \sum_{J_j^i \in (S_a^i(t) \cup S_r^i(t))} |R_j^i| + \sum_{J_j^i \in (J^i - S_c^i(t) - (S_a^i(t) \cup S_r^i(t)))} |R_j^i| \\
&\quad - \sum_{J_j^i \in (S_a^i(t) \cup S_r^i(t))} |R_j^i| - \sum_{J_j^i \in S_a^i(t)} |W_j^i| \\
&= \sum_{J_j^i \in (J^i - S_c^i(t) - (S_a^i(t) \cup S_r^i(t)))} |R_j^i| - \sum_{J_j^i \in S_a^i(t)} |W_j^i| \\
&= \sum_{J_j^i \in (J^i - S_c^i(t))} |W_j^i| - \sum_{J_j^i \in S_a^i(t)} |W_j^i| \\
&= \sum_{J_j^i \in (J^i - S_c^i(t) - S_a^i(t))} |W_j^i| + \sum_{J_j^i \in S_a^i(t)} |W_j^i| - \sum_{J_j^i \in S_a^i(t)} |W_j^i| \\
&= \sum_{J_j^i \in (J^i - S_a^i(t))} |W_j^i|
\end{aligned}
$$

∎

Equation 4.2 is also true for a more general case where a file might have multiple readers if only the sizes of distinct files are counted in both $max\_claim(i,t)$ and $alloc(i,t)$. Although we do not consider the situation in which the files created by some completed jobs are read by the subset of unscheduled jobs (i.e., $J^i - (S_a^i(t) \cup S_r^i(t))$), it does not affect the correctness of the lemma since the storage allocated to these files is counted in both $max\_claim(i,t)$ and $alloc(i,t)$.

Lemma 4.2.1 indicates that the *need matrix* is determined by the set of unscheduled jobs (i.e., $J^i - S_a^i(t)$), and thus it can be updated by subtracting $|W_j^i|$ at the time when job J_j is safely granted

```
/* DAR is invoked when job J_j^i in instance I_i
** is intended to be scheduled. r(t) is a global variable
** representing the available storage at t. t is
** monotonically increasing. |W_j^i| and |R_j^i| are
** the sizes of the write and read data sets of J_j^i in I_i.
** need(i,0) is initialized to max_claim(i,0) for I_i.
** alloc(i,t) records the amount of storage that has
** been allocated to I_i. alloc(i,0) is initialized to 0.
** Both need(i,t) and alloc(i,t) are global variables. */
DAR(I_i, J_j^i) {
    /* W_j^i and R_j^i are local variables. */
    W_j^i ← getWriteSet(J_j^i);
    if (|W_j^i| > r(t))
        /* wait until there is enough free storage*/
        return false;
    /* pretend to modify the system by assuming
    ** that J_j^i has completed.*/
    R_j^i ← getReadSet(J_j^i);
    r(t) ← r(t) − (|W_j^i| − |R_j^i|);
    alloc(i,t) ← alloc(i,t) + (|W_j^i| − |R_j^i|);
    need(i,t) ← need(i,t) − |W_j^i|; /* Lemma 4.2.1 */
    if (safetycheck(I_i))
        /* Actually, J_j^i is not completed */
        r(t) ← r(t) − |R_j^i|;
        alloc(i,t) ← alloc(i,t) + |R_j^i|;
        return true;  /* the request is safe */
    else
        /* recover the modification */
        r(t) ← r(t) + (|W_j^i| − |R_j^i|);
        alloc(i,t) ← alloc(i,t) − (|W_j^i| − |R_j^i|);
        need(i,t) ← need(i,t) + |W_j^i|;
        return false;  /* the request is unsafe */
}
```

Figure 4.2. The DAR Algorithm.

its request. We can leverage these results to design our DAR algorithm as well as its safety checking algorithm. The DAR algorithm is shown in Figures 4.2.

DAR is invoked each time a job J_j^i in instance I_i is considered for scheduling (Figure 4.2). DAR first checks if the current available storage is sufficient to satisfy the request of the job (obtained via *getWriteSet()*). If not, the job has to wait until sufficient storage is available. Otherwise, the job J_j^i is assumed to be completed, and the corresponding data structures associated with the instance I_i (i.e., $r(t)$, $alloc(i,t)$ and $need(i,t)$) are updated accordingly. Subsequently, the safety of granting the request of job J_j^i is checked using the subroutine *safetycheck()*. We assume that job J_j^i is completed before checking for the safety because we want the algorithm to be more aggressive in the safety

check. Thus, if the safety check is passed (i.e., the system is in a safe state), the request of job J_j^i is safe, but some data structures (i.e., $r(t)$ and $alloc(i,t)$) in the algorithm need to be re-adjusted since job J_j^i is actually not yet completed.

The safety checking algorithm in DAR is identical to that in the standard banker's algorithm (Chapter 2.2) when viewing each instance as a process. The algorithm iterates over all the instances in the workload and pools their allocated storage until the need matrix of the current examined instance is satisfied (i.e., a safe state) or all instances are checked but it is impossible to complete the current examined instance (i.e., an unsafe state).

The DAR algorithm correctly avoids deadlocks in workflow-based computation since the maximum claim defined in Equation (4.1) is clearly an upper bound on the storage requirements of the instance. The DAR algorithm follows the classic banker's algorithm, and thus has the same time complexity of $O(m^2)$ as the banker's algorithm for checking the safety of a request, where m is the number of instances in the workload.

As discussed earlier, DAR tends to improve the inter-instance concurrency at the expense of intra-instance concurrency. Specifically, in DAR the need matrix associated with each instance is monotonically non-increasing (possibly decreasing) as the instance proceeds; therefore, instances nearing completion will have smaller need matrices. As a result, the need matrices associated with the nearly completed instances are *easily* satisfied, and the storage they are holding is *easily* deallocated as well. The resulting deallocated storage can be gathered during the construction of the safe instance sequence to improve the possibility of safely granting requests from new instances (although their need matrices are large). Consequently, more instances can be active (i.e., inter-instance concurrency is enhanced). In contrast, the conservatively large value of the need matrix defined in the control-flow-based banker's algorithm makes it *hard* to construct safe instance sequences, thereby restricting inter-instance concurrency.

4.2.2 The DTO Algorithm

The DTO algorithm, unlike DAR, is designed to maximize the active storage utilization by trying to improve the intra-instance concurrency. To this end, we view the job rather than the instance, as in DAR, as the unit of storage allocation and deallocation and localize the safety checking of the job request to the current instance (i.e., the instance making the request).

More specifically, when deciding if a job J_j^i in instance I_i can be scheduled at moment t (see Figure 4.3), DTO first checks whether the request of the job (obtained via *getWriteSet*()) is greater

than the current available storage. If so, the job has to wait until there is sufficient storage. Otherwise, the set of jobs that have already been scheduled, $S(t)$, and the current job, J_i^r (i.e., $S(t) \cup \{J_i^r\}$) are computed, which also provides the set of jobs not yet scheduled (i.e., $\overline{S(t)} = \{J^r - (S(t) \cup \{J_i^r\})\}$), and the safety of granting the request of job J_i^r is checked.

In the safety check (Figure 4.6), it is first assumed that job J_i^r is granted the request and finished immediately and then the scheduling state (after scheduling job J_i^r) is moved by topologically ordering each job $J_k^s \in \{J^r - (S(t) \cup \{J_i^r\})\}$ (i.e., only considering all remaining jobs in instance I_r) such that

$$\forall J_m^q \in \{J^r - (S(t) \cup \{J_i^r\})\}, \quad \sum_{\substack{\forall J_k^s \in \{J^r - (S(t) \cup \{J_i^r\})\} \\ \text{and } \mathrm{idx}(J_k^s) \le \mathrm{idx}(J_m^q)}} (M_k^s - R_k^s) \le a(t) \tag{4-6}$$

Here, $a(t)$ is the available storage at moment t (after J_i^r is assumed to be finished), the initial value of $a(t)$ (i.e., for $t = 0$) is the total storage budget, J_k^s is the job that is scheduled before job J_m^q in instance I_r, $\mathrm{idx}(J_k^s)$ is the index of job J_k^s in the topologically ordered sequence. By definition, a topological ordering of a workflow DAG is a linear ordering of its jobs in which the data dependencies between the jobs are respected. The scheduling state is safe if all remaining jobs (i.e., those jobs that have not yet been scheduled in the current workflow instance) can be topologically ordered while satisfying Inequality (4-6). Otherwise, the scheduling state is unsafe. A job request is safe if the scheduling state is safe after the job request is granted.

For example, in Figure 4.1(b), if at the moment $T_{1,1,0}$ is and $J_{1,1,0}$ is the first job to be finished and the available storage is 20 units. At this point, $J_{1,1,0}$ is making a request of 5. For checking the safety of this request, DYRS first assumes $J_{1,1,0}$ is be finished. Then, $a(t)$ is $20+5=25$. Based on Inequality (4-6), we topologically order the remaining jobs $J_{1,2,0}$, $J_{1,3,0}$, $J_{2,1,0}$ and $J_{2,2,0}$ by following an order $J_{1,2,0} \rightarrow J_{1,3,0} \rightarrow J_{2,1,0} \rightarrow J_{2,2,0}$. Then, we have

1. $25 > 5$ ($J_{1,2,0}$ can be finished)
2. $25 - 5 + 5 = 25 > 5$ ($J_{1,3,0}$ can be finished)
3. $25 - 5 + 5 - 5 + 5 = 25 > 5$ ($J_{2,1,0}$ can be finished)

So, we can topologically order the remaining jobs and the request of $J_{1,1,0}$ is safe.

With regard to the safety check, we concern is the existence of such a safe job sequence, which indicates that given a storage budget, there is at least one deadlock-free job schedule if the order specified in the safe job sequence is followed. However, given an arbitrary DAG and a storage budget, determining the existence of such a safe job sequence is generally believed to be a NP-complete problem [67]. Therefore, instead of looking for an optimal algorithm to find the safe job

```
/* DTO is invoked when J_j^i in I_i is intended
** to be scheduled at moment t. Be aware that t is
** monotonically increasing. S_s(t)^i is the set of
** jobs in I_i that have been scheduled before t.
** r(t) is a global variable representing the
** available storage at t.
** |W_j^i| is the size of the write data set of J_j in I_i */
DTO (I_i, J_j^i) {
    /* local variables: W_j^i, S_s^i(t), S_c^i(t) and S_a^i(t) */
    W_j^i ← getWriteSet(J_j^i);
    if (|W_j^i| > r(t))
        /* wait until there is enough free storage*/
        return false;
    S_c^i(t) ← getCompletedJobs(I_i);
    S_a^i(t) ← getActiveJobs(I_i);
    S_s^i(t) ← S_c^i(t) ∪ S_a^i(t); /* see Table 4.1 */
    if (safetycheck(S_s^i(t), J_j^i))
        r(t) ← r(t) − |W_j^i|;
        /* the request of J_j^i is safe */
        return true;
    else
        /* the request of J_j^i is unsafe */
        return false;
}
```

Figure 4.3. The DTO Algorithm

sequence, we use some heuristics to topologically order the jobs that satisfy Inequality (4.4) (see the topological order algorithm in Figure 4.5).

In the safety checking algorithm *topological_order ()*, shown in Figure 4.5, is our proposed function to topologically order the remaining jobs so that each ordered job will have sufficient storage to satisfy its request (as determined by Inequality (4.4)). The algorithm searches the (remaining) dataflow DAG in breadth-first order and expands the nodes whose parent nodes have all been expanded (i.e., the ready jobs). A node is expanded if its request can be satisfied by the amount of storage computed in the right side of Inequality (4.4). Given the available storage, if all the remaining jobs can be ordered, the algorithm returns *true*, indicating a safe scheduling state; otherwise, the algorithm returns *false*. Again, a job request is safe if the scheduling state is safe.

Since our algorithm is built on top of breadth-first traversal, the complexity of the algorithm for checking the safety of a request is $O(n^2)$, where n is the number of jobs in the instance. Our breadth-first-based algorithm is different from the situation in the banker's algorithm where the nodes (i.e., processes) are independent without data dependencies.

It can be shown that DTO can avoid deadlock among multiple concurrent workflow instances

```
/* checking if scheduling J_j^i is safe at moment t.
** |R_j^i| is the size of read data set of J_j in I_i */
safetycheck(S_s^i(t), J_j^i) {
    /* R_j^i, W_j^i, d and C^i are local variables. */
    d ← r(t);
    R_j^i ← getReadSet(J_j^i); /* get read set */
    W_j^i ← getWriteSet(J_j^i); /* get write set */
    d ← d − |W_j^i| + |R_j^i|;
    /* assume C^i is the set of completed jobs of I_i */
    C^i ← S_s^i(t) ∪ {J_j^i}; /*assume J_j^i is finished*/
    /* topological_order is the function that
    ** topologically orders the unscheduled jobs
    ** to satisfy Inequality (4.4) */
    return topological_order(d, C^i, i);
}
```

Figure 4.4. The Safety Checking Algorithm in DTO: The algorithm is performed to find out if a job scheduling in instance I_i is in a safe state.

and also improve the intra-instance concurrency. The correctness of this algorithm is not difficult to understand since via (informal) inductive arguments, we can see that there always exists at least one active instance and the state is safe. The DTO algorithm, as with the banker's algorithm and DAR, only allows for transitions into safe states. Therefore, a non-safe (i.e., deadlock possible) state is never entered.

As discussed earlier, DTO tends to improve intra-instance concurrency at the expense of inter-instance concurrency. Specifically, DTO is biased to grant resource requests to instances nearing completion (i.e., intra-instance), rather than start (or admit) new instances (i.e., inter-instance). When resources are tight, the resource requests for new instances tend to be larger than the resources available. However, the resource requests of admitted instances, which are monotonically non-increasing due to the topological order, might be satisfiable under the same constraints.

4.2.3 Summary

In this section we described two deadlock avoidance algorithms, DAR and DTO, for workflow-based computation when storage resources are constrained. Both algorithms are based on the well-known banker's algorithm. However, our algorithms leverage the dataflow information and make a distinction between active and inactive resources to minimize the makespan, which is different from the previous work where improving resource utilization was usually the goal.

With DAR the maximum claim associated with each instance is dynamically computed by using the dataflow information to sum the resource requirements of all the remaining jobs (i.e., those jobs

```
/* given J^i and r(t), if we can topologically order the jobs in J^i − C^i
** to satisfy Inequality (4.4) at t.
** DAG^i is the dataflow DAG of I_i, which is globally accessible. */
topological_order(d, C^i, i) {
    /* Q is a local variable that keeps the set of all remaining
    ** unmarked nodes with no incoming edges (i.e., ready jobs) */
    Q ← φ; /* local variable */
    for (each J^i_d ∈ C^i) do
        mark J^i_d as Done in DAG^i;
        /* search in breadth-first order */
        for (each node J^i_k ∈ (J^i − C^i) with an edge e from J^i_p to J^i_k) do
            remove edge e from the DAG^i;
            if (J^i_k is a ready job)
                put J^i_k into the end of Q;
    while (T ≠ φ) do
        J^i_r ← Q.pop(); /* the first ready job */
        W^i_r ← getWriteSet(J^i_r);
        R^i_r ← getReadSet(J^i_r);
        if (|W^i_r| > d)
            return false; /* scheduling is unsafe */
        mark J^i_r as Done; /* satisfying Inequality (4.4) */
        d ← d − |W^i_r| + |R^i_r|;
        C^i ← C^i ∪ {J^i_r};
        /* search in breadth-first order */
        for (each node J^i_k ∈ (J^i − C^i) with an edge e from J^i_r to J^i_k) do
            remove edge e from the DAG^i
            if (J^i_k is a ready job)
                put J^i_k into the end of Q;
    return true; /* scheduling is safe */
}
```

Figure 4.5. The topological_order algorithm in the DTO algorithm

that have not yet been finished). The DTO algorithm exploits the dataflow information to topologically order the remaining jobs in the current instance when checking for safety (i.e., a specific order of job completion that is within a resource budget). Both algorithms try to maximize the active storage utilization by either improving the inter-instance concurrency (DAR) or improving the intra-instance concurrency (DTO).

4.3 Deadlock Avoidance Batch Scheduler

We describe a *deadlock avoidance batch scheduler*, which provides a natural way to integrate batch schedulers with deadlock avoidance algorithms. Specifically, we introduce a component known as a *deadlock avoider* that implements both DAR and DTO and we integrate it with a *priority-based* batch scheduler for checking the safety of job requests. The resulting scheduler architecture is shown in Figure 4.6.

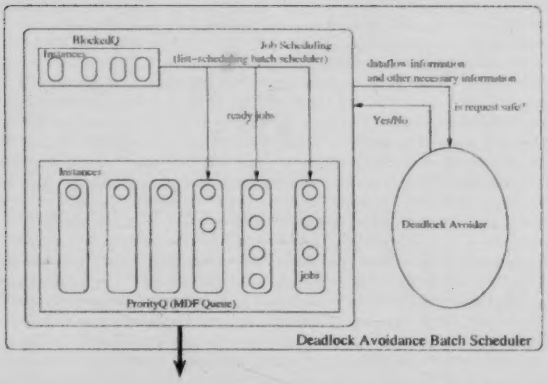


Figure 4.6. Deadlock Avoidance Batch Scheduler. MDF stands for Most Done Job First, an Instance Scheduling Policy.

4.3.1 The Priority-based Batch Scheduler

In a priority-based batch scheduler the instances are assigned priorities and ordered in a list in decreasing magnitude of priority (i.e., *PriorityQ* in Figure 4.6). In addition, a priority sub-queue is also maintained for each instance to prioritize its ready jobs. The processors are allocated to instances first and then to the jobs in the selected instance according to a higher-priority-first policy. Ties are broken randomly [27].

Depending on evaluation metrics, instances can be prioritized based on different strategies; for example, *First Come First Serve* (FCFS) for fairness, *Shortest Job First* (SJF) for minimizing the mean response time and so forth. Although these strategies are popular and always used as the default in most batch schedulers for processor allocation (e.g., Maui/Moab [54] and IBM's LoadLeveler [53]), they are not entirely suitable when the total makespan is the performance concern and the instances can be preemptive at some particular points when jobs are finished. To meet our demands, we propose a simple strategy, called *Most-Done Job First* (MDF) to prioritize the instances. In this strategy, the instance with the largest number of completed jobs has the highest priority (ties are broken by the instance ids assigned by the scheduler when the instances enter the system). The motivation for this strategy is the hope that the instances who have completed most of the jobs will finish as soon as possible, thereby minimizing the storage contention by freeing storage resources.

If no data dependence constraints are violated, the job priorities can also be determined in a variety of ways such as *Highest Level First* (HLF) [17], *Longest Path* (LP) [17], *Longest Processing*

State	Event	Action
1	One or more new instances arrive	Accepting incoming instances and initializing $PriorityQ$
2	One or more jobs are finished	Resolving the dataflow dependencies and updating some data structures to record the changes of the free and allocated storage
3	No active instances, but $PriorityQ$ is not empty	Triggering the batch scheduling

Table 4.4. Scheduling States, Events and Corresponding Actions

Time (LPT) [32, 55] and *Critical Path* (CP) [40]. Moreover, the priority computation can be either at submission time, *static scheduling*; at runtime, *dynamic scheduling* [57, 86, 104]; or by a combination of both. In our job scheduling we adopt HLF, a simple strategy not requiring the processor information, but an effective heuristic to approximately speed up the computation along the critical path. The notion of level is the sum of computation costs of all of the nodes along the longest path from the node to be scheduled to the sink node. The motivation for this heuristic is the hope that it will minimize the instance execution time and thus release the held storage as soon as possible.

In addition to the priority queue, there is another important queue called *BlockedQ*, which maintains the jobs for each instance whose data dependencies have not yet been resolved. If a job's data dependencies have been resolved, it becomes a ready job and enters the corresponding job priority queue.

4.3.2 Integration with Deadlock Avoider

The deadlock avoider is integrated with the batch scheduler using inter-process communication. The batch scheduler records the request of each job and is responsible for scheduling the instances and jobs by allocating resources to them (i.e., processors and storage; storage being our concern). More specifically, in order to safely grant the request of a job, the scheduler will ask the deadlock avoider if the request is safe or not. The deadlock avoider will answer "Yes" or "No" by running DAR or DTO for the safety check.

The batch scheduler is event-driven: there are three major states to process three kinds of scheduling events at moment t. The processing states and their corresponding trigger events as well as the major actions in each stage are shown in Table 4.4, and the algorithms corresponding to each state are described in Figures 4.7, 4.8 and 4.9, respectively.

The scheduling algorithm for State 1 (new instance arrival) is simple; it accepts (by *accept_instance()*) an incoming instance at moment t and assigns an identifier to it (i.e., i for the instance I_i in the algo-

```
/* State 1: accepting incoming instances at moment t.*/
i ← 0;
do
    I_i ← accept_instance();
    for (each job J_j^i ∈ I_i) do
        if (J_j^i's data dependency has been resolved)
            insert J_j^i into PriorityQ;
        else
            put J_j^i into BlockQ;
    i++;
until (all incoming instances are accepted)
```

Figure 4.7. Algorithm for Dealing with New Instance Arrivals

rithm). Then the jobs in the instance are checked and either inserted into *PriorityQ* for scheduling or put into *BlockQ* awaiting dependency resolution (each job is also assigned a job id, but this is not shown in the algorithm). This procedure is repeated for all incoming instances at moment t until all are processed.

State 2 (job(s) finished) deals with the situation where a set of jobs are completed at moment t. It first updates the available storage (the allocated storage to the instance to which the completed job belongs) by adding up (deducting) the released storage of the completed job and then, if possible, frees any jobs in *BlockQ* that data depend on the completed job placing them in the *PriorityQ* queue (using *dep_resolver()*). This procedure is repeated for all completed jobs at moment t until all are processed. Finally, it checks *PriorityQ* based on some particular scheduling strategies (MDF and HLF in our case) to select the instance and the job to grant the storage requests if *deadlock_avoider()* returns true.

State 3 (scheduling required) is necessary to avoid the situation when there are no running jobs but the computation has not been finished (i.e., *PriorityQ* is not empty). In this case the State 3 algorithm is triggered, acting the same as the scheduling part of State 2 except that when no job's request is granted after scheduling, it reports the storage is insufficient to complete the computation. In both State 2 and State 3, the job whose request has been (safely) granted is removed from the corresponding instance's job queue. The algorithm of *deadlock_avoider* is fairly simple. Its pseudo code is shown in Figure 4.10. Depending on the parameter alg, either DAR or DTO is invoked for the safety check whenever a job J_j^i in some instance I_i is selected to be scheduled.

```
/* State 2: One or more jobs are finished at moment t*/
do
    /* when a job J_j^i in Instance I_i finishes at moment t*/
    ** the available storage at t and the allocated storage to I_i
    ** are first updated. r(t) and alloc(i,t) are global variables. */
    r(t) ← r(t) + |R_j^i|;
    alloc(i,t) ← alloc(i,t) − |R_j^i|;
    /*Check BlockQ to insert those jobs (i.e., H^i) whose data
    ** dependency has been resolved into PriorityQ.*/
    H^i ← dep_solver(i,j);
    for (each job J_j^i ∈ H^i) do
        insert J_j^i into PriorityQ;
until (all completed jobs are processed)
/* Scheduling Part: finally checks PriorityQ
** to select instance and job to grant the storage
** requests if no potential deadlock could be incurred.
** PriorityQ[i] contains all jobs of instance I_i.*/
for (each instance I_i in PriorityQ) do
    for (each job J_j^i in PriorityQ[i]) do
        if (deadlock_avoider(I_i, J_j^i, alg))
            schedule job J_j^i;
            remove job J_j^i from PriorityQ[i];
```

Figure 4.8. Algorithm for Dealing with Job Completions

4.4 Active-Instance-Aware Admission Control

Too little concurrency will limit performance. Chapter 4.2 has already discussed efficiency from the point of view of maximizing concurrency by introducing new algorithms (i.e., DAR and DTO) that use dataflow information to dynamically compute maximum resource needs. However, too much concurrency of the wrong kind can also have a detrimental effect. Specifically, too many admitted instances that are blocked on unsatisfiable resource requests leads to inactive resource allocation. In this section we discuss the role of instance admission control, also based on dataflow information, to further improve makespans.

The following is a simple example to illustrate the problems that can occur when there is no admission control (see Figure 4.11). Suppose that initially we have 13 storage units and each file has a unit size. `Job A` in each instance thus require 3 storage units. The requests from all the `Job A`s (one `Job A` per instance) of the four workflow instances can be safely granted, and the four instances are admitted to execution, achieving the maximum *degree of concurrency* (DOC) (Figure 4.11(a)). After one of the `Job A`s is finished, say `Job A` in `WI1`, it is safe to grant the request of `Job B` in `WI1` since there is one storage unit left. Unfortunately, in the following computation no progress can be made on `WI2`, `WI3` and `WI4` since no storage left for the output files of `Job`

```
/* State 3: No active instances, but PriorityQ is not empty
** at moment t check PriorityQ to select instance and job to
** grant the storage requests if no potential deadlock could
** be incurred. PriorityQ[i] contains all jobs of instance I_i.*/
for (each instance I_i in PriorityQ) do
    for (each job J_j^i in PriorityQ[i]) do
        if (deadlock_avoider(I_i, J_j^i, alg))
            schedule job J_j^i;
            remove job J_j^i from PriorityQ[i];
if (no job is scheduled)
    print "the storage is insufficient to
           complete the computation.";
```

Figure 4.9. Algorithm for Triggering Batch Scheduling

```
/* depending on alg, either DAR or DTO is invoked to
** check the safety of the request of J_j^i in I_i. */
deadlock_avoider(I_i, J_j^i, alg) {
    switch(alg) {
        case DAR:
            return DAR(I_i, J_j^i;)
        case DTO:
            return DTO(I_i, J_j^i;)
        default:
            print "Unknown Deadlock Avoidance Algorithm";
            return false;
    }
}
```

Figure 4.10. The Deadlock Avoider Algorithm

`B`, `Job C` and `Job D` in these three instances. As a result the storage they hold becomes inactive until `WI1` is finished. When storage is allocated but is inactive, there is less storage to be allocated to other non-blocked jobs, which tends to reduce concurrency, thus likely increasing makespan.

The problem of too much and too little concurrency is analogous to the problems faced by operating systems in controlling the degree of multiprogramming. However, storage resources are not preemptable (unlike memory, for instance), and dataflow information can be valuable when making admission control decisions. For example, using knowledge of the Fork&Join workflow shape of the instance, an admission control algorithm can reduce, the degree of concurrency of the system in the short term to achieve better concurrency in the long term. In Figure 4.11(b) only two instances (i.e., the right hand side) are initially admitted, and 7 storage units (out of a total of 13 units) are left unallocated. In the short term, this is less than the four instances admitted and 1 storage unit unallocated in Figure 4.11(a). However, after the Job As are completed, the degree of concurrency in Figure 4.11(b) increases to six (i.e., Jobs `B`, `C` and `D` of `WI1` and `WI2`) since there will be enough

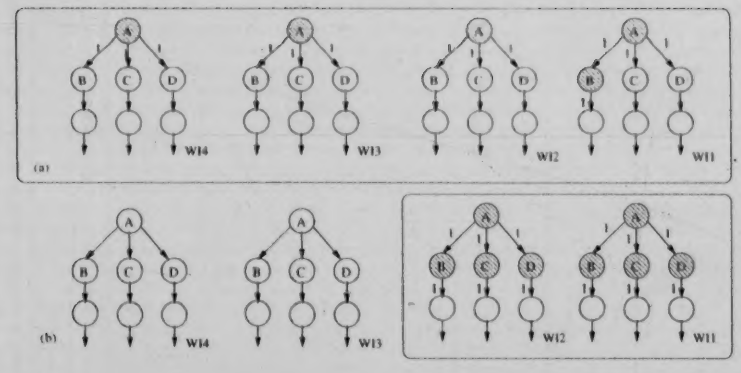


Figure 4.11. An example illustrating that allocating storage without admission control may incur poor performance.

storage for those jobs. In contrast, as discussed above for Figure 4.11(a), only `Job B` of `WI1` can run after `Job A` of `WI1` finishes. Although Figure 4.11 is a specific example where admission control has a benefit, we will show that the benefits are more general (Chapter 4.5.7).

4.4.1 The Admission Control Algorithm

In this section, we propose a simple *active-instance-aware admission control* algorithm (*Instance Admission Control* (IAC), for short) and integrate it with the deadlock avoider to further minimize the amount of inactive storage. The parameter space for admission control is large, and we show significant benefits for our algorithm, but we do not perform an exhaustive study of admission control. Our main point is that admission control can be valuable in addition to the DAR and DTO algorithms.

The basic idea of the IAC algorithm is to use the dataflow information to estimate the average number of active instances so that each such instance could be ensured of a *moderate* amount of storage to maximize its job concurrency. However, due to the characteristics of the workload (e.g., workflow shape) the definition of "moderate" is not easy to determine. Our simple heuristic is to compute the average degree of job concurrency for the workflow shape (Equation (4.5), Table 4.5) and, subsequently, the average storage requirement of an instance (Equation (4.6), Table 4.5). Then by simply dividing the total amount of storage available by the average storage requirement of each instance, we have a target average number of concurrent instances for the system (Equation (4.7), Table 4.5).

More specifically, the algorithm first estimates the average degree of job concurrency (i.e., avg_doc)

symbol	*meaning*
avg_fs	the average file size
B	the given storage budget
$total\_files$	the total number of files in the workflow DAG
$total\_jobs$	the total number of nodes in the workflow DAG
max_doc	the maximum degree of job concurrency
min_doc	the minimum degree of job concurrency
avg_doc	the average degree of job concurrency
avg_st	the average storage held by each instance
avg_ic	the average degree of instance concurrency
avg_jst	the average job service time
avg_cp	the average length of critical path
mk	the total makespan of the computation

Table 4.5. Notation Used in Instance Admission Control

for the workflow shape as follows (some notation used is shown in Table 4.5):

$$avg\_doc \approx \frac{min\_doc + max\_doc}{2} \tag{4.5}$$

It then approximates the average storage held by each instance (i.e., avg_st) using:

$$avg\_st \approx 2 \cdot avg\_doc \cdot \frac{total\_files}{total\_jobs} \cdot avg\_fs \tag{4.6}$$

Here, the factor 2 means that we count both input and output files for each job.

Finally, given storage budget B, the algorithm estimates the number of concurrent instances as:

$$avg\_ic \approx \frac{B}{avg\_st} \tag{4.7}$$

The advantage of Equation (4.7) is that avg_ic is simply a function of min_doc and max_doc of the workflow shape. It does not need other information related to instances such as job service time and critical path.

Although Equation (4.7) simplifies the estimation, it also introduces inaccuracy, and thus it may result in poor performance in some special cases. The inaccuracy of this estimation is caused by the assumption implied by Equation (4.5) that the degree of job concurrency is changed smoothly during the instance computation.

From Equation (4.5) we can see that the key point in our algorithm is the computation of min_doc and max_doc of the workflow. This can be achieved by knowing the dataflow DAG of the workflow. min_doc is relatively easy to compute since our workflow model assumes a single entry node. In our experiments, max_doc is easily computed for the relatively straightforward workflow shapes.

```
/* the extended deadlock_avoider. J_j^i in I_i is intended
** to schedule at moment t. The running instances at t
** are those instances that have active jobs at t.
** S is the id set of running instance at t and avg_ic
** is computed by Equation (4.7) (see Chapter 4.4.1). */
deadlock_avoider_ext(I_i, J_j^i, alg) {
    S ← getActiveInstanceIdSet();
    if (i ∉ S)
        if (|S| ≥ avg_ic) /* |S| is the size of S */
            return false;
    return deadlock_avoider(I_i, J_j^i, alg);
}
```

Figure 4.12. The Extended Deadlock Avoider after Integrating with Instance Admission Control

For the general case of arbitrary DAGs we note that max_doc can be computed via the *Dilworth Decomposition* algorithm [24, 95]. The Dilworth algorithm actually computes the maximum antichain for the DAG, which can be used as an upper bound on the maximum degree of concurrency.

Based on the estimated value of avg_ic, the deadlock avoider is extended (Figure 4.12) to add the admission control (called from the code in Figure 4.8).

Before checking the safety of a job's request, the algorithm first obtains (by using *getActiveInstanceIdSet()*) the *id* set of running instances S at moment t (i.e., those instances that have active jobs) and controls the instance admission as follows:

1. If the selected instance I_i is not running (i.e., I_i has no running jobs), processing its jobs might increase the number of running instances. Thus, if the number of running instances (i.e.,$|S|$) has reached the upper bound (i.e., avg_ic), the algorithm returns false, denying the instance admission.

2. If the selected instance I_i is running, processing its jobs does not increase the number of running instances, and thus the jobs in the selected instance can be processed directly by invoking the original deadlock avoider (i.e., DAR or DTO).

4.5 Simulation Results

We evaluated the performance of the proposed algorithms through a simulation-based study. The simulator in Chapter 3 was extended to include the deadlock avoider, which integrates both DAR and DTO for deadlock avoidance. On each occasion, a multiple-instance workload, together with the control-flow DAG as well as the estimated job service time and data file sizes of each constituent

job, was submitted to the scheduler. As discussed earlier, the scheduler obtains the detected dataflow from WaFS. The instance scheduling strategy is based on Most Done Job First (MDF) while the job scheduling strategy is based on Highest Level First (HLF) [17].

4.5.1 Methodology

In the simulation study, we have four main methodology axes:

1. *Benchmark Workloads:* The benchmark workloads are characterized by the workflow shape, workflow shape parameter, job characteristics and so on.
2. *Reference Algorithms:* Our reference algorithms are the standard banker's algorithm, Lang's algorithm [60] and a deadlock detection algorithm.
3. *Simulated Platforms:* We assume that an unbounded number of homogeneous nodes but limited total storage budget is available.
4. *Instance Admission Control:* No instance admission control is assumed in our baseline strategies in Chapter 4.5.5 and 4.5.6.

4.5.1.1 Benchmark Workloads

We still use the two representative structures, Fork&Join and Lattice as well as their special case, Pipeline, as our experimental workflow shapes. In our experiments, for instances of all workflows, we assume that the job service time (JST) is uniformly distributed, and every output data file has a size with a uniform distribution. In addition, each workload contains 100 instances and all instances are assumed to arrive (at the batch scheduler) at the same time since this situation is both the common and worst case in terms of storage contention. The characteristics of the benchmark workloads are in Table 3.2 in Chapter 3.

4.5.1.2 Reference Algorithms

- **Banker's Algorithm :** Since, without dataflow information, storage space, in general, cannot be safely reclaimed until the end of each instance, we thus use the aggregate storage requirements of *all* jobs in instance as its global maximum claim for the control-flow-based banker's algorithm.

- **Long's Algorithm:** Long's algorithm [60] is a more recently proposed deadlock avoidance algorithm which demonstrates an advantage over the standard banker's algorithm in terms of resource utilization. However, our empirical results show that this algorithm, except for some special cases (Figure 4.28(b)), never outperforms the banker's algorithm in terms of makespan when applying it to our benchmark Pipeline workloads (Chapter 4.5.5.1 and 4.5.5.3).

- **Deadlock Detection:** To evaluate the difference between our deadlock avoidance algorithms and other kinds of deadlock resolution algorithms, we compare DAR and DTO with deadlock detection. The parameter space for deadlock detection is (non-exhaustively) explored. The basic idea of the deadlock detection algorithm is to have the batch scheduler detect the deadlock at the earliest time and then use a variety of strategies to compute the amount of storage that needs to be released after a deadlock has been detected, select the victims and reallocate the released storage to recover from the deadlock. Through a parameter space study we found that the deadlock detection algorithm has the best overall performance when half of the storage budget is released after a deadlock has been detected and the victim instances are selected based on the Least-Done Job First (LDF) criteria. We denote the deadlock detection algorithm as *Det(0.50Byt,LDF)* and use it as a reference algorithm to evaluate our deadlock avoidance algorithms, DAR and DTO (Chapter 4.5.6).

4.5.1.3 Simulated Platforms

In the study we further assume that an unbounded number of homogeneous computational nodes are available so that the maximum DOC is never constrained by the hardware except for the storage. The total storage budget in each experiment is limited.

4.5.1.4 Instance Admission Control

As will be discussed in Chapter 4.5.7, instance admission control can be beneficial in reducing the makespan of some workloads (e.g., Lattice and Pipeline). We have not included admission control in the baseline strategies of Chapter 4.5.5 and 4.5.6 because the actual admission control policy of Chapter 4.5.7 is simplistic (which makes it unworthy of setting a standard). Also, we wish to limit the number of policy parameters and elements when considering the impacts of workflow and workload characteristics.

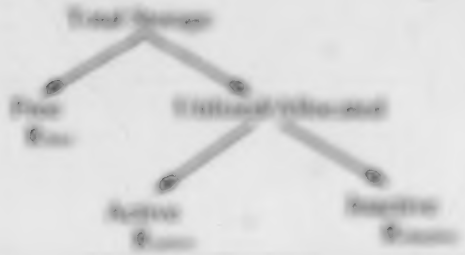


Figure 6.10: Classification of Storage Resource Utilization

6.5.2 Performance Metrics

To evaluate the algorithms we have two primary metrics: makespan and active/inactive storage utilization.

1. Makespan: Makespan is the amount of time it takes to complete all the jobs of a workload, from the submission of the first job to the completion of the last job (Figure 6.11). We use it to measure the algorithm's performance.

2. Active and Inactive Storage Utilization: The classification of the storage resources is shown in Figure 6.10. We use the following way to define the active and inactive storage utilization (Equation 6.8):

$$\theta_{sto} = \frac{\sum_{i=0}^{makespan} \delta_{sto}(i)\Delta t}{makespan \cdot total\_storage} \qquad (6.8)$$

where $\delta_{sto}(i)$ is the total amount of storage in the state of sto (active, inactive) at moment i. $\theta_{sto} = 0 \sim 1$ ($\theta_{active} + \theta_{inactive} = 1$) specifies the ratio of the storage. Here θ_{active} / $\theta_{inactive}$ is the traditional storage resource utilization.

6.5.3 Organization

The simulation studies are organized as follows: In Chapter 6.6.1 we study the parameters of our algorithms to avoid workload characteristics and show how our algorithms can tuned. Also we compare our deadline algorithms in terms of reduced makespans and high active storage utilization. Then, in Chapter 6.6.2 we compare our algorithms with the reference deadline unaware algorithm and show the improvements under reduced SAR and CPU consumption for proactive algorithms. Finally, we evaluate the performance benefits of resource utilization control in Chapter 6.6.3.

Deviation	DAR (%)	DTO (%)	Banker's (%)	Lang's (%)	Detection (%)	Overall (%)
< 10%	98.7	98.7	93.6	71.1	79.8	92.9
< 20%	100	100	100	91.1	93.6	98.6
< 30%	100	100	100	100	96.9	99.5
< 40%	100	100	100	100	99.4	99.9
< 50%	100	100	100	100	100	100
	No std. deviation bars			With std. deviation bars		

Table 4.8. The Distribution of the Standard Deviations of the Makespan Data in Our Simulations

4.5.4 Data Points and Standard Deviation

Each data point in the makespan and storage usage graphs (see y-axis) is averaged over 10 runs by changing the random number seed in the simulator for each run. Important exceptions to the multiple run methodology are the trace data used for Figures 4.18, 4.19, 4.23, 4.24 and 4.32, which are based on a single, representative run.

Again, to measure the distribution of the set of 10 values for each makespan data point, we compute the standard deviations of the data points in Chapters 4.5.5 through 4.5.7 (see Table 4.6). We found that, overall, 92.9% of standard deviations are less than 10% of the data point's value. Specifically, for the DAR and DTO algorithms 98.7% of the standard deviations were less than 10% of the data point's value. In the case of the banker's algorithm, 92.9% of the standard deviations were less than 10% of the data point's value. Therefore, for clarity of presentation we have omitted standard deviation bars on the graphs for DAR, DTO and the banker's algorithm data points. A similar presentation strategy was used in Chapter 3.4.2.

Lastly in our simulation, for all graphs where the x-axis represents storage units, the leftmost, starting point on the x-axis is based on the largest maximum claim of the banker's algorithm for all workflow instances. It is not possible to run DAR and the banker's algorithm with storage budgets of less than this maximum claim value. For example, in Figure 4.14(a), the x-axis begins at 250 units because the largest maximum claim of the banker's algorithm for all 100 workflow instances is 203 units.

4.5.5 Sensitivity to Workload Characteristics

In this section we present some simulation results on the sensitivities of our proposed algorithms to workload characteristics and show, in a variety of cases, how our proposed algorithms outperform the banker's algorithm (for both Fork&Join and Lattice workloads) and Lang's algorithm (for

Pipeline workloads, since Lang's algorithm cannot effectively process a workload that has a general structured workflow graph) in terms of makespan and active storage utilization.

For instances of all the benchmark workflows, their job service times and file sizes are assumed to be over-estimated and uniformly distributed on [500, 1000] time units and [1, 10] storage units, respectively. In our experiments we also assume that each job-created file has only one reader. However, to reflect the reality, the sensitivities of the proposed algorithms to multiple-reader access patterns is also studied (Chapter 4.5.5.5).

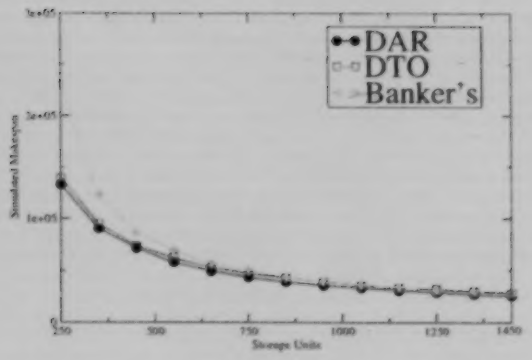
In general, it is difficult to explain the performance differences between deadlock avoidance algorithms since their abilities to maximize the resource utilization may be different or, sometimes, incomparable. Roughly speaking, in our studies the performance of the compared deadlock avoidance algorithms largely depends on their capability to make a distinction between the active and inactive storage (reflected in their safety check of a job's request), which are related to the following workflow characteristics:

1. *Workflow shape:* the structure of the workflow, e.g., Pipeline.
2. *Workflow shape parameters:* the parameters that describe the workflow with a particular shape.
3. *Workflow size:* the scale of the workflow specified by the total number of nodes (jobs).
4. *Job characteristics:* the job service time and data file sizes associated with each job
5. *File Access Pattern:* how a job-created file is read, either by a single job or by multiple jobs.
6. *Total storage budget:* the given amount of storage that can be used by the entire computation.

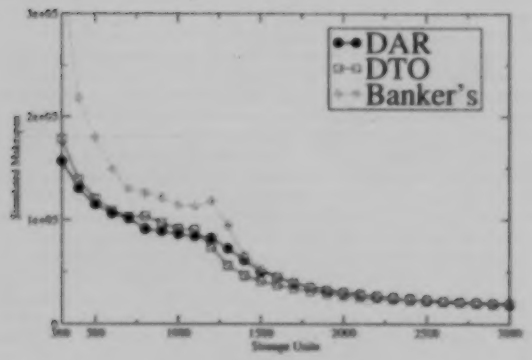
The relative performance between the compared algorithms will change as these factors change. We investigate the impact of these factors by varying the storage budget.

4.5.5.1 Sensitivity to Workflow Shapes: Performance Changes Depend on the Shapes

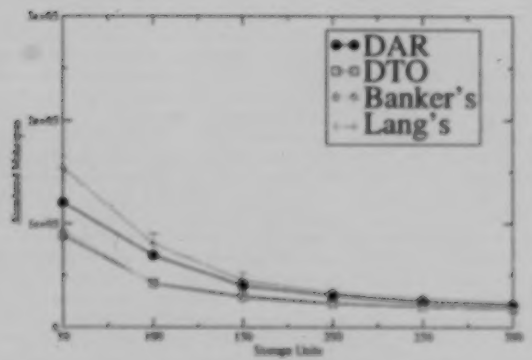
The experiments in this section are intended to show how the compared algorithms are sensitive to the workflow shapes. For both the examined Fork&Join and Lattice workloads, DAR and DTO are consistently better than the banker's algorithm. For the Pipeline workload, DAR and DTO are consistently better than Lang's algorithm, and DTO and the banker's algorithm are competitive in terms of showing the best performance. In addition, we also compared the storage utilization of the algorithms and found that in general our proposed algorithms have high active storage utilization.



(a) Fork&Join (3×8)



(b) Lattice (4×6)



(c) Pipeline (5-stage)

Figure 4.14. Impacts of Workflow Shape on the Makespans of the Compared Algorithms when the Workflow Sizes are Small

The relative performance between DAR and DTO also depends on the workflow shapes; neither one algorithm can consistently outperform the other across all the examined workflow shapes.

Comparison with the Banker's Algorithm and Lang's Algorithm (The comparisons with the reference deadlock detection algorithm are detailed in Chapter 4.5.6) Figure 4.14 shows how DAR and DTO are consistently better than the banker's algorithm and Lang's algorithm for the benchmark workloads of Fork&Join (3×8), Lattice (4×6) and Pipeline (5-stage). It also shows how DTO and the banker's algorithm are competitive in terms of having the best performance for the Pipeline workload.

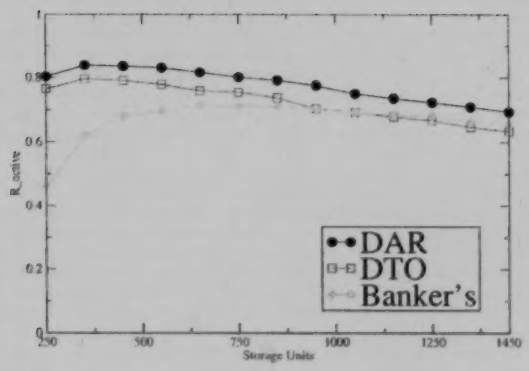
For the Fork&Join, our proposed algorithms demonstrate better performance than the banker's algorithm since they have higher resource utilization. Due to the large intra-instance concurrency, the allocated storage resources to the Fork&Join workload can be effectively used. This is evidenced by the high active storage utilizations of all compared algorithms shown in Figure 4.15(a). Thus, improving the resource utilization can directly improve the active storage utilization as well as the performance for the Fork&Join (3×8).

The same observation can be obtained from the Lattice workload. As shown in Figure 4.16, both our proposed algorithms outperform the banker's algorithm by improving the active storage utilization. However, due to the low intra-instance concurrency (i.e., high data dependency in the Lattice DAG), the active storage utilizations of all compared algorithms for the Lattice workload are lower than those for the Fork&Join workload.

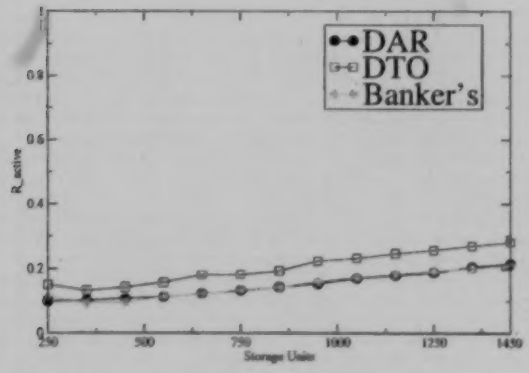
In addition, we can see from Figure 4.16 that as the storage budget increases from 300 units to a certain value such as 1200 units, the active storage utilization for each algorithm decreases while the inactive storage utilization increases, but the overall performance remains largely unchanged or becomes slightly better. At first, these two observations seem contradictory, but, in actual fact, they are not.

On one hand, as the storage increases, more and more instances can be safely granted the resources and admitted to execution. However, due to the storage constraints most of them become blocked, holding inactive storage. The number of blocked jobs, and the inactive storage utilization, increase monotonically as the storage budget increases. On the other hand, the large number of blocked jobs also indicates that a large amount of work has been finished. As a result, the overall performance might be improved or not changed.

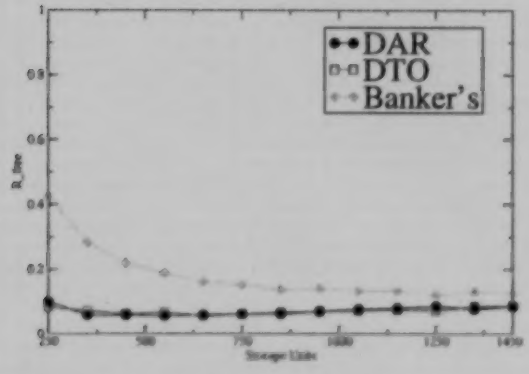
When the storage increases over a certain value, all the instances in the Lattice workload can be admitted. Thereafter, increasing storage undoubtedly minimizes the inactive storage and reduces the makespan. Due to the high data dependency inside the Lattice workflow, the available storage might



(a) Active Storage Utilization

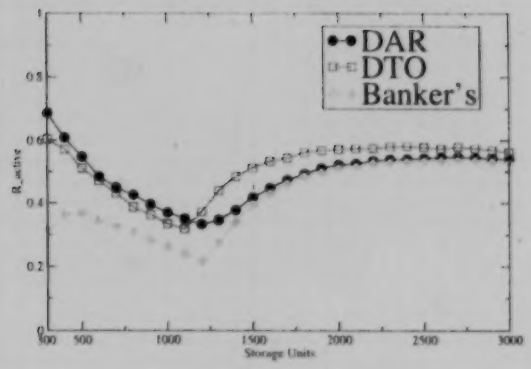


(b) Inactive Storage Utilization

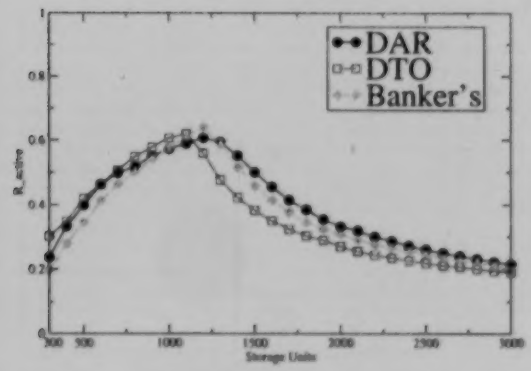


(c) Free Storage Ratio

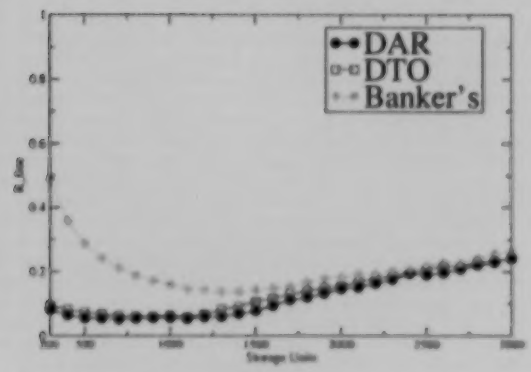
Figure 4.15. How the storage is used by each of compared algorithms for Fork&Join (3×8).



(a) Active Storage Utilization



(b) Inactive Storage Utilization



(c) Free Storage Ratio

Figure 4.16. How the storage is used by each of the compared algorithms for Lattice (4×6).

not be efficiently used. Thus, the compared algorithms for the Lattice have relatively higher free storage ratio than for the Fork&Join, especially when the storage increases over 1500 units (compare Figures 4.15(c) and 4.16(c)).

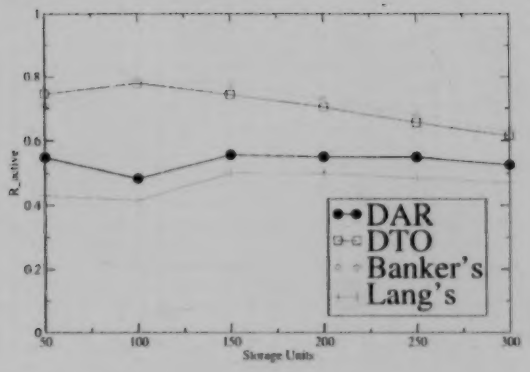
Figure 4.14(c) shows how DAR and DTO outperform Lang's algorithm and DTO competes with the banker's algorithm to have the best performance for a 5-stage Pipeline workload. The low performance of Lang's algorithm shows that high resource utilization does not always imply high performance for a workflow-based workload and making the distinction between active and inactive storage is important for reducing the makespan.

As we know, Lang's algorithm has the highest resource utilization among the algorithms being compared (our experimental results validate the claim of Lang [60]; see Figure 4.17(c)). However, the high resource utilization does not result in high performance. Rather, Lang's algorithm suffers from the worst performance. This is because improving resource utilization also implies increasing inactive storage utilization, which has adverse effects on the performance (Figure 4.17). This explanation is confirmed by comparing DAR and DTO with Lang's algorithm, where both DAR and DTO show lower resource utilization but better performance.

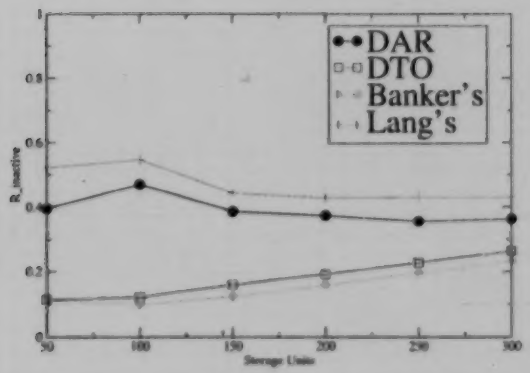
Another interesting observation is that the banker's algorithm, which has the lowest resource utilization, is comparable to the best performance (DTO) for the 5-stage Pipeline workload. We think the reason for this is that due to the Pipeline structure, the defined global maximum claim can act as a good instance admission control for the efficient use of the storage. In other words, the maximum claim is relatively small so that a sufficient number of instances can be admitted to execution, yet it is big enough to prevent too many instances from being admitted, thereby minimizing the storage contention and lowering the inactive storage (Figure 4.17(b)).

The Relative Performance between DAR and DTO: Figure 4.14 also shows how, depending on the workflow shape, the relative performance between DAR and DTO change. Roughly speaking, as the workflow shape is varied and the request of the first job in each instance is reduced, the relative performance of DAR to DTO becomes worse. However, when the storage is highly constrained and the data dependencies inside the workflow are high, DAR is slightly better than DTO.

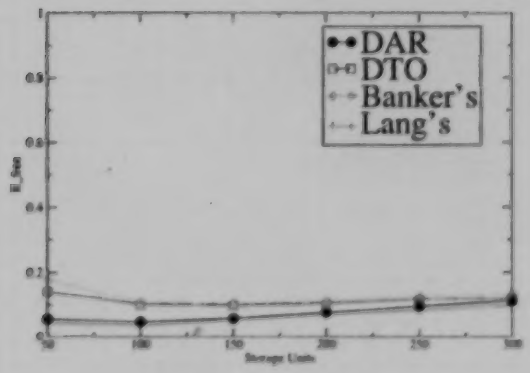
More specifically, as the number of edges going out of the first node from the workflow DAGs decrease (i.e., the request of the first job is reduced) from 8 (Fork&Join (3×8)) to 2 (Lattice (4×6)) and further to 1 (Pipeline (5-stage)), the relative performance of DAR gradually becomes worse, from being consistently better than DTO for the Fork&Join (Figure 4.14(a)) to consistently worse than DTO for the Pipeline (Figure 4.14(c)). As for the Lattice workload, DAR exhibits marginally better performance than DTO when the storage budget is low and slightly worse performance when



(a) Active Storage Utilization

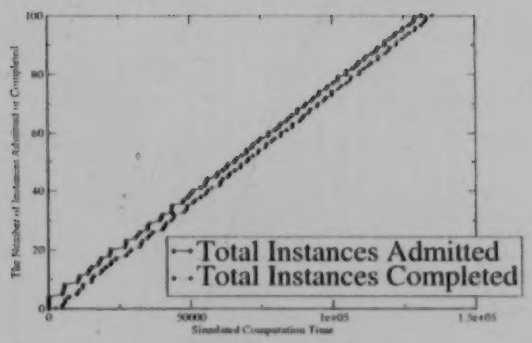


(b) Inactive Storage Utilization

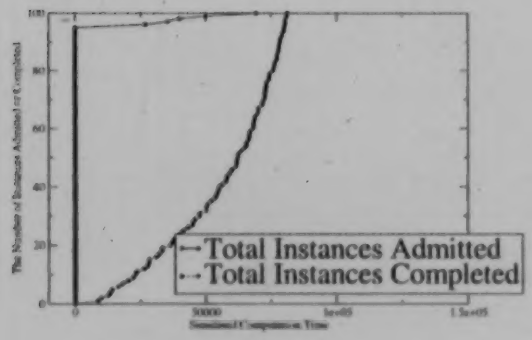


(c) Free Storage Ratio

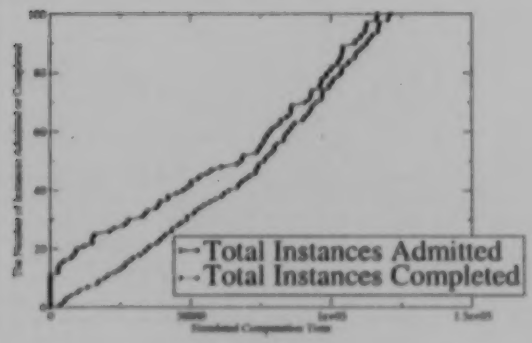
Figure 4.17. How the storage is used by each of the compared algorithms for Pipeline (5-stage).



(a) DAR: Fork&Join (3×8)

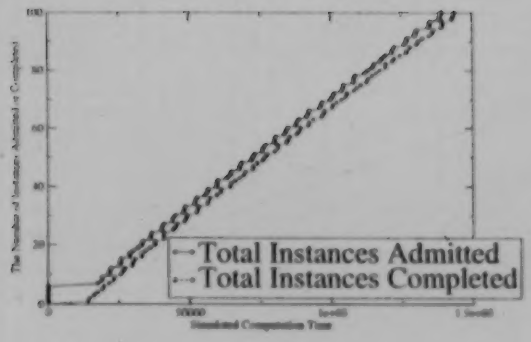


(b) DAR: Lattice (4×6)

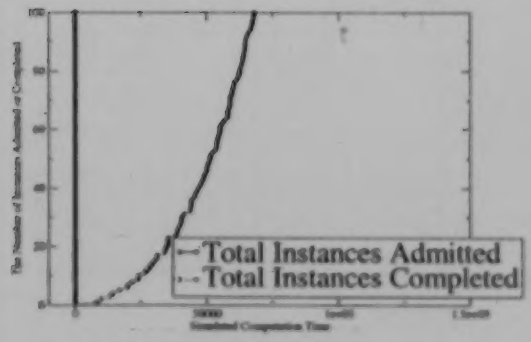


(c) DAR: Pipeline (5-stage)

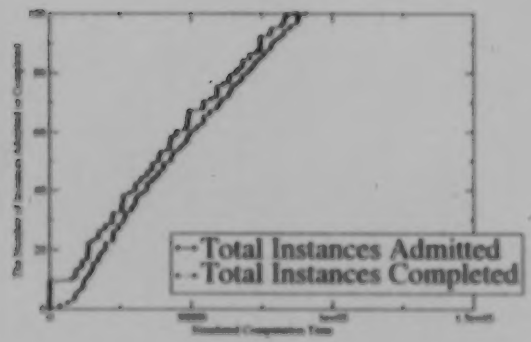
Figure 4.18. Execution Traces (DAR): The total number of workflow instances that are admitted and completed by DAR as the computation proceeds, given the storage budget of 250 units for the Fork&Join (3×8), 1200 units for the Lattice (4×6), and 50 units for the Pipeline (5-stage).



(a) DTO: Fork&Join (3 × 8)



(b) DTO: Lattice (4 × 6)



(c) DTO: Pipeline (5-stage)

Figure 4.19. Execution Traces (DTO): The total number of workflow instances that are admitted and completed by DTO as the computation proceeds, given the storage budget of 250 units for the Fork&Join (3 × 8), 1200 units for the Lattice (4 × 6), and 50 units for the Pipeline (5-stage).

	Stage	Fan-out
Fork&Join	12	2
	6	4
	3	8*
	2	12
	Height	**Width**
Lattice	2	12
	4	6*
	3	8

Table 4.7. Investigated Workflow Shape Parameters: The total number of jobs is fixed as 24 for Lattice and 26 for Fork&Join (two extra nodes for the source and sink). * indicates the shape parameters that were studied in the previous experiments.

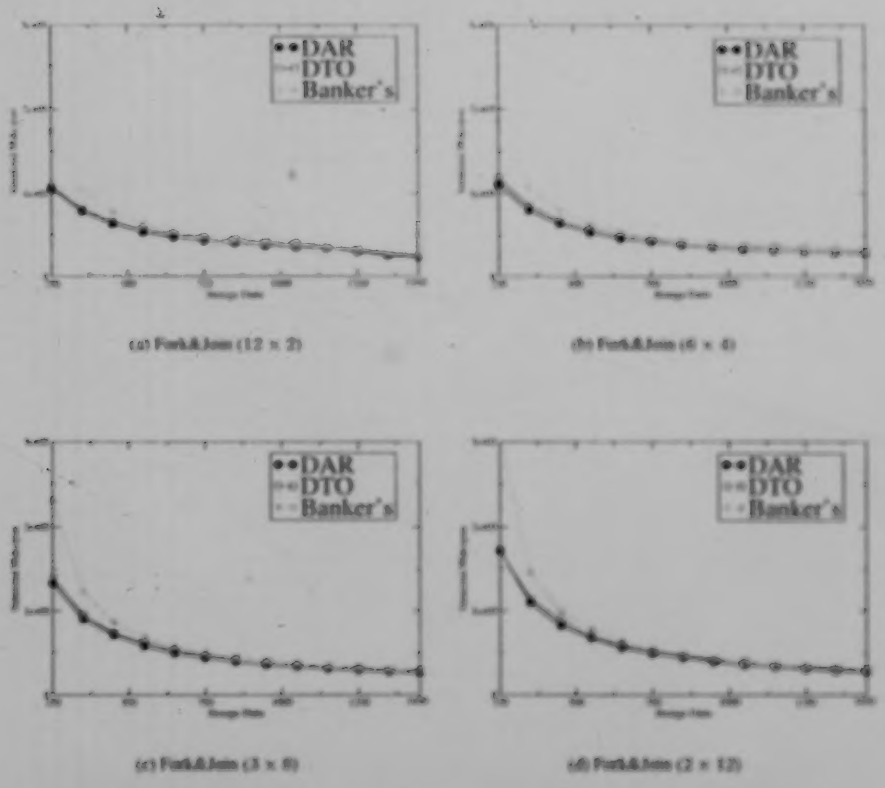
the storage budget is increased over a certain value (1100 units in Figure 4.14(b)).

These results are not surprising since DAR tends to maximize inter-instance concurrency; then, as the request of the first job is reduced, DAR is biased to admit more instances to execution. Consequently, the storage allocated to each instance is minimized and thus the *instance completion rate* of DAR is decreased. The instance completion rate is defined by how many instances have been completed during a given period of time. Clearly, the higher the instance completion rate, the lower the makespan.

We can validate this explanation by observing Figures 4.18 and 4.19, where DAR has a lower instance completion rate than DTO for both the Lattice and Pipeline. However, for the Fork&Join, due to its large fan-out factor, admitting instances becomes difficult and thus relatively more resources can be left to the already admitted instances to increase their completion rates. From Figures 4.18(a) and 4.19(a), we can see that the performance of DAR is largely determined by the instance completion rate during the early stage of the computation, which is slightly higher than that of DTO.

When the storage budget is highly limited and data dependency inside the workflow is high, DAR is slightly better than DTO (see Figure 4.14(b) where the storage budget is less than 1000 units). This, again, is not surprising since, due to the large value of the maximum claim and the highly limited resources, it becomes difficult for DAR to construct a safe instance sequence for checking the safety of job requests. As a result, it admits fewer instances than DTO, leaving more storage to the admitted instances, thereby improving the instance completion rate. These results are more pronounced when the workflow size is enlarged (see Chapter 4.5.5.3).

In summary, depending on the workflow shape and available storage budget, neither DAR nor DTO will consistently outperform the other.



(a) Fork&Join (12 × 2)

(b) Fork&Join (6 × 4)

(c) Fork&Join (3 × 8)

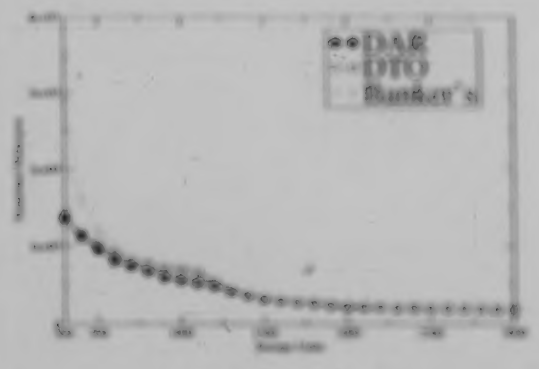
(d) Fork&Join (2 × 12)

Figure 4.20. Impacts of Workflow Shape Parameters on the Makespans of the Compared Algorithms for the Fork&Join (26 jobs): Note that Figure 4.20(c) is identical to Figure 4.14(a).

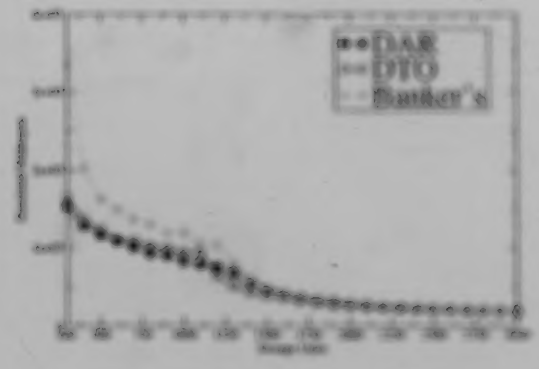
4.5.5.2 Insensitivity to Workflow Shape Parameters: Performance Changes Are Not Sensitive

The following experiments show how the compared algorithms are insensitive to the workflow shape parameters in terms relative performance. To this end, we focus on the Fork&Join and Lattice workflows, fix their total number of jobs (i.e., the workflow size) and vary their shape parameters (see Table 4.7).

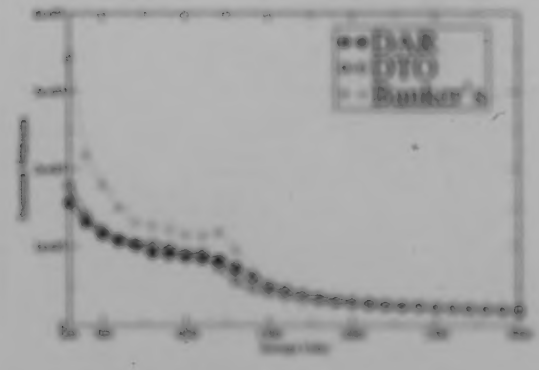
Figure 4.20 shows that for all studied parameters DAR and DTO perform better than the banker's algorithm for the Fork&Join and, as the fan-out factor increases, the performance of all compared algorithms is slightly degraded. These results are expected since, compared with DAR and DTO, the banker's algorithm is conservative in resource utilization, and as the fan-out factor increases, the



(a) Lattice (2×10)

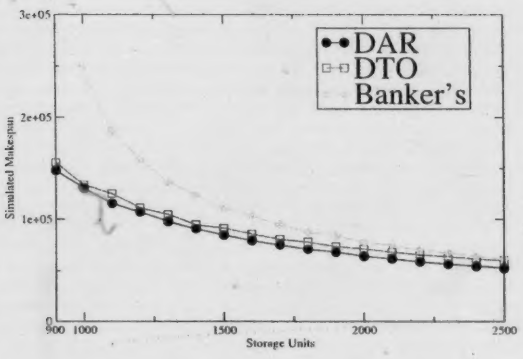


(b) Lattice (6×6)

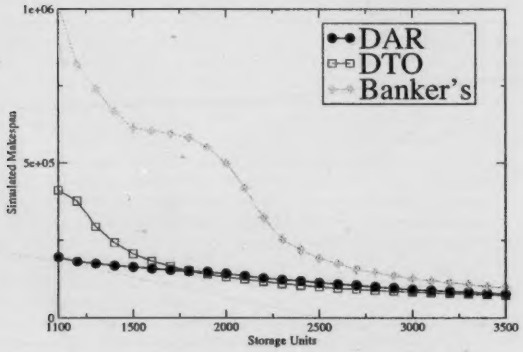


(c) Lattice (9×6)

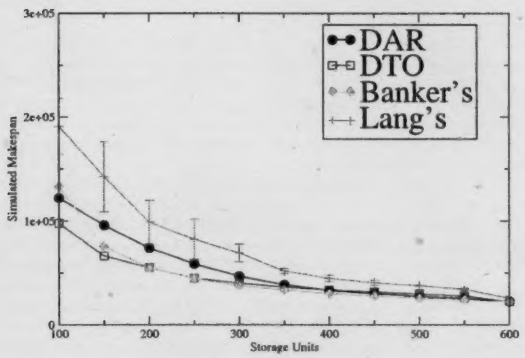
Figure 4.21. Impact of Workflow Shape Parameters on the Makespans of the Compared Algorithms for Lattice ($|\lambda|$ jobs). Note that Figure 4.21(c) is identical to Figure 4.14(b) with different scales along the y-axis.



(a) Fork&Join (3×32)

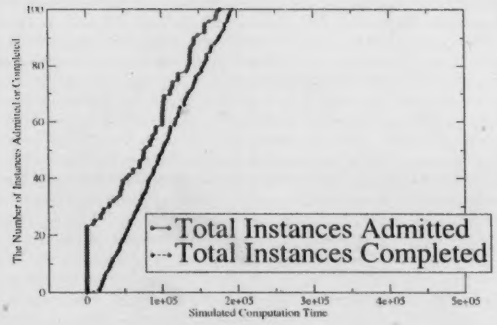


(b) Lattice (8×12)

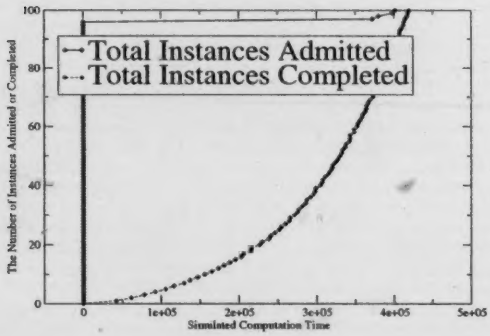


(c) Pipeline (10-stage)

Figure 4.22. Impacts of Workflow Size on the Makespans of the Compared Algorithms when the Workflow Sizes become Large


The Number of Instances Admitted or Completed
Simulated Computation Time
Total Instances Admitted
Total Instances Completed

(a) DAR: Lattice (8×12)

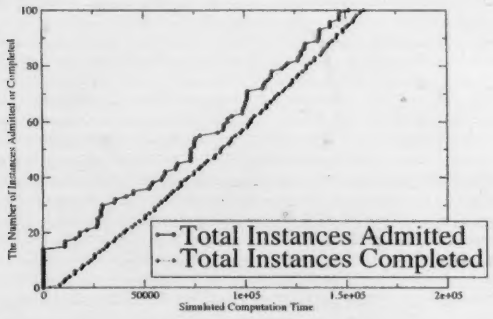

The Number of Instances Admitted or Completed
Simulated Computation Time
Total Instances Admitted
Total Instances Completed

(b) DTO: Lattice (8×12)

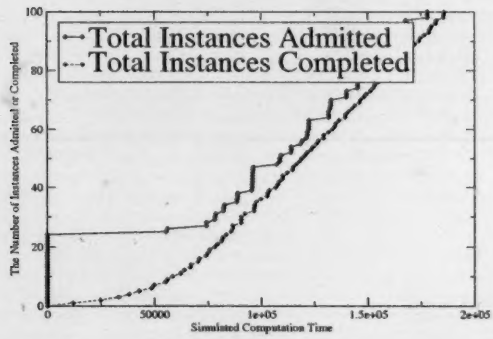
Figure 4.23. Execution Traces (Lattice (8×12)): The total number of workflow instances that are admitted and completed by DAR and DTO as the computation proceeds, given a storage budget of 1100 units for the Lattice (8×12).

matically degraded when the storage budget is low, which is different from that for the Pipeline workload whose degradation is relatively small. Such a performance sensitivity is not surprising since the banker's algorithm is based on the control-flow information, and the defined global maximum claim is determined by the total *file sizes* in the workflow instead of by jobs. Except for the Pipeline, the total file sizes increase quickly as the workflow size becomes large. As a result, the large value of the maximum claim significantly reduces the storage resource utilization in the banker's algorithm, leading to poor performance.

As the workflow size increases, the performance differences between DAR and DTO are slightly enlarged for the Fork&Join (3×32) (see Figure 4.22(a)) but significantly enlarged for the Lattice ($8 \times$



(a) DAR: Lattice (4×6)



(b) DTO: Lattice (4×6)

Figure 4.24. Execution Traces (Lattice (4×6)): The total number of workflow instances that are admitted and completed by DAR and DTO as the computation proceeds, given a storage budget of 300 units for the Lattice (4×6).

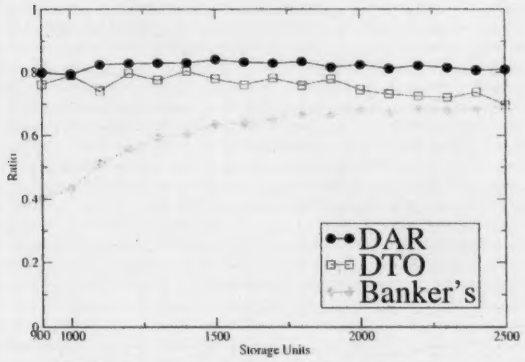
12) (up to 50%), especially when the storage budget is low (see Figure 4.22(b)). We attribute these results to the increased differences between the numbers of instances admitted by both algorithms at the beginning of the computation as well as the effects of the workflow shapes. The number of instances admitted at the beginning of the computation is our concern because a large amount of storage might be held as inactive storage for a long time (possibly starting from the scratch), thereby compromising the performance significantly. For the Fork&Join, although the difference between the numbers increases as the workflow size becomes large, it is not as large as that for the Lattice due to the effects of the large fan-out factor. Therefore, the performance difference between DAR and DTO is relatively small for the Fork&Join.

For the Lattice (8×12) the large performance difference between DAR and DTO mostly results from the large difference between the numbers of instances admitted by DTO and DAR at the beginning of the computation (i.e., 96 vs. 23; see Figure 4.23). DTO admits more instances and thus has much less storage left for the admitted instances than DAR, resulting in more blocked instances/jobs and inactive storage, which further results in a low instance completion rate (Figure 4.23). However, the difference in instance admission between DTO and DAR is not large for the Lattice (4×6): only 24 instances are initially admitted by DTO and 16 instances by DAR (i.e., 24 vs. 16; see Figure 4.24). The difference between these numbers is primarily determined by the localized maximum claim of DAR, which depends on the workflow size.

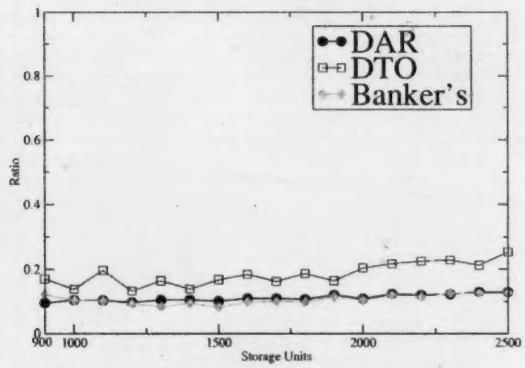
As shown in Figure 4.22(c), the relative performance difference between DAR and DTO remains largely unchanged as the number of Pipeline stages increases. However, the performance differences of the proposed algorithms and Lang's algorithm are relatively enlarged. The performance of Lang's algorithm is degraded as the Pipeline size increases, thereby enlarging its performance differences between DAR and DTO. These results are not surprising since Lang's algorithm has the highest resource utilization. It is thus relatively easy to admit and block more instances than both DAR and DTO during the computation, resulting in a large amount of inactive storage, especially when the number of Pipeline stages increases.

To further validate our explanations, we also show how the storage resources are used by each algorithm for the examined workloads in Figures 4.25, 4.26 and 4.27. As expected, our deadlock algorithms have higher active resource utilization overall than both the banker's algorithm and Lang's algorithm (when the storage is highly constrained for the Pipeline; less than 250 units). Moreover, compared to the small workflow size, DAR and DTO exhibit many more advantages for improving active resource utilization when the workflow size is large, which is consistent with our makespan results shown in Figure 4.22.

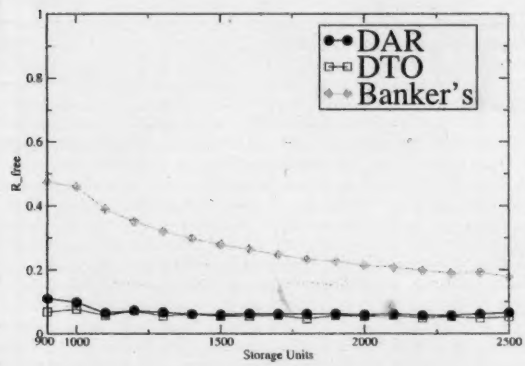
From these experiments, we can conclude that the relative performance between the compared algorithms is largely unchanged, but their performance differences are enlarged as the workflow size increases, especially for the Lattice workload when storage is highly constrained.



(a) Fork&Join (3×32) Active Storage Utilization

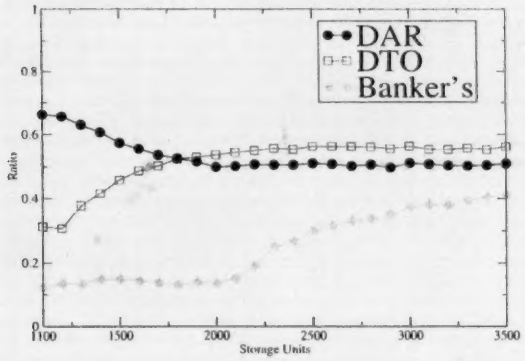


(b) Fork&Join (3×32) Inactive Storage Utilization

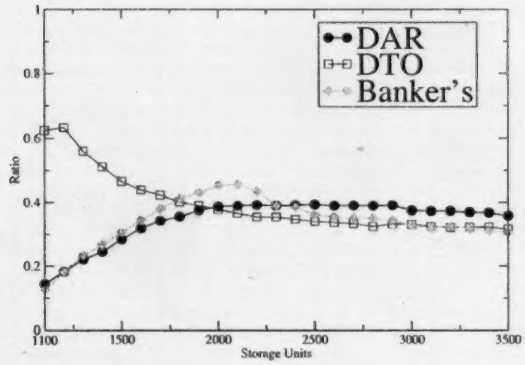


(c) Fork&Join (3×32) Free Ratio

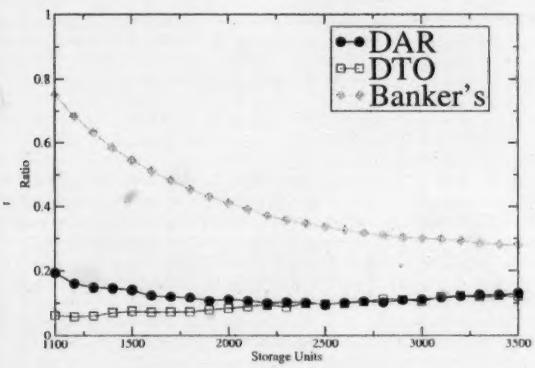
Figure 4.25. How the storage is used by each of the compared algorithms for the Fork&Join (3×32).



(a) Lattice (8 × 12) Active Storage Utilization

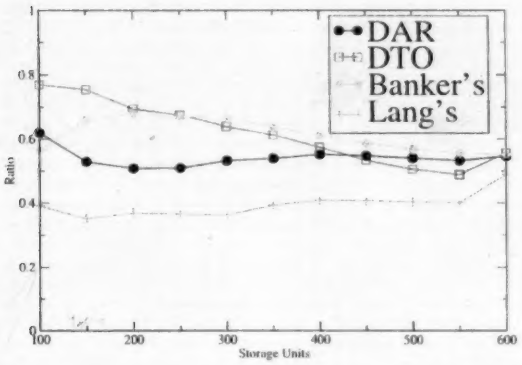


(b) Lattice (8 × 12) Inactive Storage Utilization

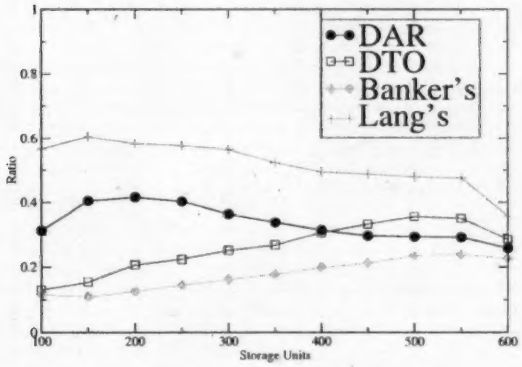


(c) Lattice (8 × 12) Free Storage Ratio

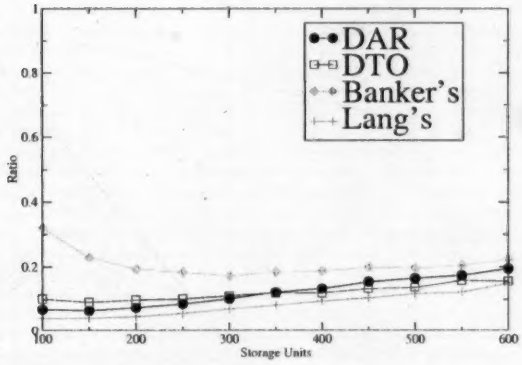
Figure 4.26. How the storage is used by each of the compared algorithms for the Lattice (8×12).



(a) Pipeline (10-stage) Active Storage Utilization

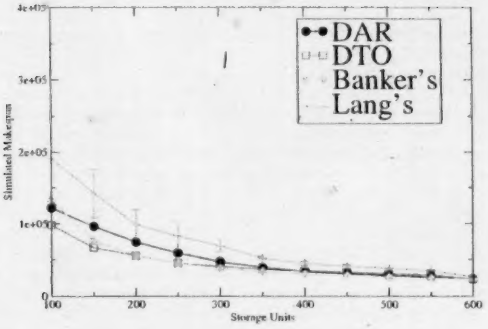


(b) Pipeline (10-stage) Inactive Storage Utilization

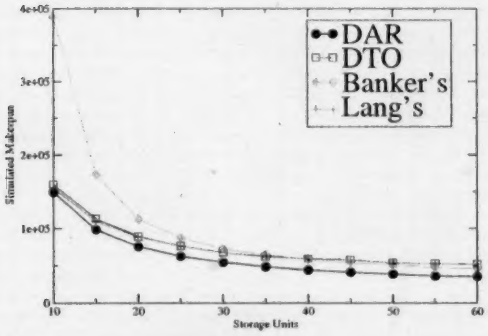


(c) Pipeline (10-stage) Free Storage Ratio

Figure 4.27. How the storage is used by each of the compared algorithms for the Pipeline (10-stage).



(a) Pipeline (10-stage), JST(500, 1000), FS(1,10)



(b) Pipeline (10-stage), JST(500, 1000), FS(1,1)

Figure 4.28. Impacts of File Size Distribution Parameters on the Makespans of the Compared Algorithms: Figure 4.28(a) is the same as Figure 4.22(c).

4.5.5.4 Insensitivity to Job Characteristics: Performance Remains Largely Unchanged

In the following experiments we show how the relative performance of the compared algorithms is insensitive to the job characteristics. Jobs are characterized by their job service times and input/output file sizes.

To this end, we conducted the same experiments as those in Chapter 4.5.5.3 while changing the distribution parameters of the job service time and file size. For the job service time, in addition to [500, 1000], we also considered the uniform distribution ranges of [10, 1000] and [800, 1000]. The job service time indicates how long the allocated storage is held by each job. Similarly, in addition

to [1, 10], the uniform distribution parameters of the file size were changed to [1, 1] (i.e., unit size) and [1, 50]. Our experimental results show that except for the Pipeline workload with unit file size, the relative performance between the compared algorithms is not changed.[1]

Figure 4.28(b) shows the simulation results for the Pipeline workload with unit file size when the job service time is uniformly distributed on [500, 1000] time units. We found that the relative performances of DAR and Lang's algorithm are dramatically improved (even over DTO), while the banker's algorithm is degraded, which is different from the observations in the Pipeline with variable file sizes (see Figure 4.28(a)). We attribute these changes to the unit file size and corresponding dynamically decreased maximum claims for DAR and Lang's algorithm. Because all the files have unit size and each instance is scheduled based on MDF, the released storage of the completed jobs can be *easily* reused by the already admitted instances in DAR and Lang's algorithm instead of admitting more new blocked instances (maximum claims are dynamically decreased and thus the safe instance sequence is easy to construct). This is different from the Pipeline workload with variable file sizes, where new instances are relatively easily admitted, thereby incurring inactive storage.

For the banker's algorithm, given the unit file size, the constant maximum claim is relatively large, and the allocated storage to each instance is either one or two units, which makes it difficult to build the safe sequence during the safety check. As a result, the instances that could otherwise be admitted when the file sizes are varied to reduce the total makespan might be blocked, leading to poor performance.

4.5.5.5 Sensitivity to File Access Patterns: Relative Performance is Altered

In the following experiments we show how the relative performance between the compared algorithms is altered when the file access patterns are changed from single reader to multiple readers. Specifically, a file created by a job might have multiple readers (i.e.,a multiple-reader access pattern), instead of a single one (i.e., a single-reader access pattern) as was assumed in earlier experiments.

To this end, we conduct an experiment by using a Fork&Join (3×32) workflow in which a single output file of the first job is read by all its child jobs. We denote the workflow as Fork&Join$^+$ (3×32). An example of such a workflow in practice is the *Proteome Analyst* (PA) web service [91] described in Chapter 2, where the homologs found by the BLAST job are fed into different pipelines to classify the proteome.

[1] Since these results are not significant, we thus do not show their corresponding figures.

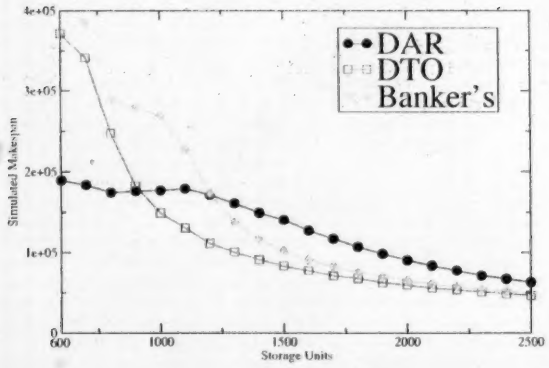


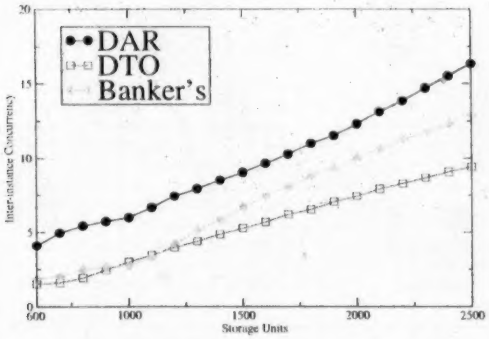
Figure 4.29. Makespan Comparisons: The algorithms are compared for the Fork&Join (3×32) workload with a file access pattern such that a single output file of the first job is read by all its child jobs (i.e., Fork&Join$^+$ (3×32)).

Figure 4.29 shows that DAR is not consistently better than DTO and the banker's algorithm, which differs from the single-reader access pattern. The relative performance between DAR and DTO is quite similar to the situation in the Lattice, but the performance differences after DTO outperforms DAR are relatively large.

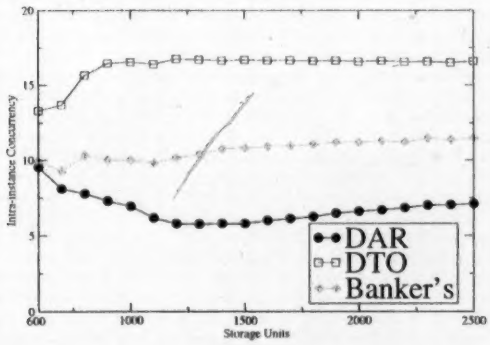
Since the single file of the first job is read by all its child jobs, the effects of the large fan-out factor are lessened. As a result, the inter-instance concurrency of DAR is increased, but the storage allocated to each instance is reduced and thus lowers the intra-instance concurrency notably.

To validate our explanations, we profiled some runtime information of the compared algorithms that are presented in Figures 4.30 and 4.31, where the inter and intra-instance concurrencies of each algorithm for both access patterns (i.e., multiple readers and single reader) are plotted for comparison. From these figures, we can see that for DAR, compared to the single reader situation, more than half of the intra-instance concurrency is reduced for the multiple-reader situation. Consequently, the instance completion rate in the multiple-reader situation is slowed down, leading to degraded performance.

In addition to the degraded performance of DAR, another reason for the banker's algorithm to outperform DAR in the multiple-reader situation is that the reduced maximum claim defined in the banker's algorithm can play a dual role in improving the performance. On one hand, it can improve the storage utilization (i.e., minimization of the free storage) and thus allow more instances to run concurrently (compare Figures 4.30(a) and 4.31(a)). On the other hand, it can act as a controller to prevent the inter-instance concurrency from being overly increased, leaving the storage resources



(a) Inter-instance Concurrency: Multiple Readers

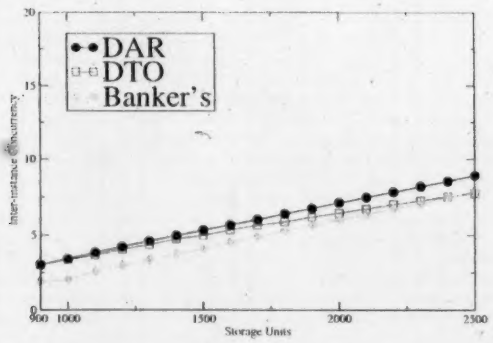


(b) Intra-instance Concurrency: Multiple Readers

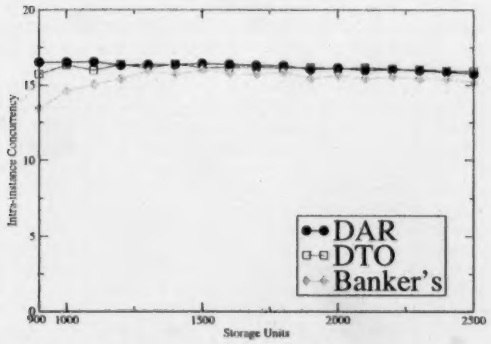
Figure 4.30. Concurrency Comparisons: The algorithms are compared for the Fork&Join$^+$ (3×32) with a multiple-reader access pattern.

available for maximizing the intra-instance concurrency. From Figure 4.32, where the execution traces of each algorithm under a given storage budget of 1400 units are plotted, we can see that the instance completion rate achieved by the banker's algorithm is higher than that achieved by DAR, which can lead to better performance.

In contrast to DAR, DTO is consistently better than the banker's algorithm. From Figure 4.30(b), we can observe that the performance advantages of DTO are primarily from its large intra-instance concurrency. This observation is expected since as the effects of the fan-out factor diminish, the value of the need matrix associated with each instance in DTO becomes small. Consequently, compared to the single-reader situation, requests from the jobs in the admitted instances can be safely



(a) Inter-instance Concurrency: Single Reader



(b) Intra-instance Concurrency: Single Reader

Figure 4.31. Concurrency Comparisons: The algorithms are compared for the Fork&Join (3×32) with a single-reader access pattern.

granted much more easily.

DAR outperforms DTO when the storage resources are highly constrained. This is expected since although the effects of the large fan-out factor are lessened, the relatively large value of the maximum claim defined in DAR (compared to that defined in DTO) still limits the number of admitted instances. Thus, DAR achieves a better performance, which is similar to the situation in the Lattice (Chapter 4.5.5.3).

Overall, our proposed algorithms are better than the banker's algorithm for the Fork&Join$^+$ (3×32) with this new file access pattern, especially when the storage budget is low. This again demonstrates that by leveraging the dataflow information, we can maximize the active storage and

reduce the makespan for workflow scheduling.

4.5.5.6 Summary

In summary, based on the simulation results we have the following conclusions:

1. Overall, both our proposed algorithms, DAR and DTO, are better than the banker's algorithm and Lang's algorithm for the benchmark workloads with different workflow shapes, workflow shape parameters, workflow sizes, job service time and file size distributions. Specifically, for different file access patterns in Fork&Join, DTO consistently outperforms the banker's algorithm. These results demonstrate that dataflow information is valuable in designing deadlock avoidance algorithms to gain performance benefits for workflow scheduling.

2. In different situations, either DAR or DTO will be the best algorithm. Roughly speaking, as the workflow shape is varied from Fork&Join to Pipeline and the request of the first job in each instance is reduced, the relative performance of DAR to DTO becomes worse (i.e., from consistently better than DTO for Fork&Join to consistently worse than DTO for Pipeline). Specifically, if the storage is highly constrained and the data dependency inside the workflow is high, DAR outperforms DTO. Otherwise, DTO shows some performance advantages over DAR.

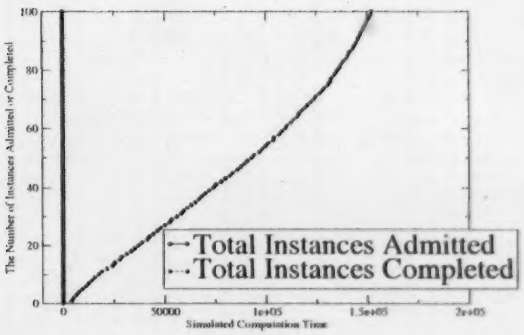
3. Compared with the control-flow-based banker's algorithm, both proposed dataflow-based algorithms are less sensitive to the workflow shape parameters and job characteristics.

4. Unexpectedly, Lang's improvements to the banker's algorithm in terms of total storage utilization do not always result in improved makespans since much of the utilized storage is inactive utilization, which compromises the makespan. This result shows that making a distinction between active and inactive storage is important to minimizing makespan.

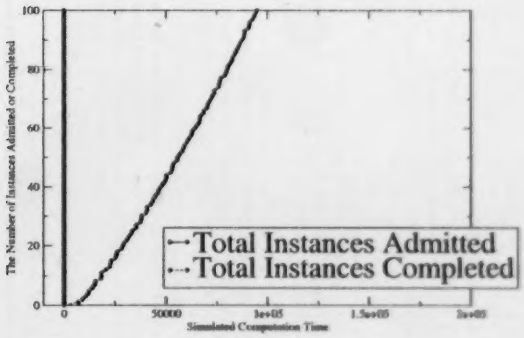
4.5.6 Comparison with Deadlock Detection

In the following set of experiments we show how DAR and DTO in most cases outperform the reference deadlock detection algorithm $Det(0.50Bgt\_LDF)$ (see Chapter 4.5.1.2).

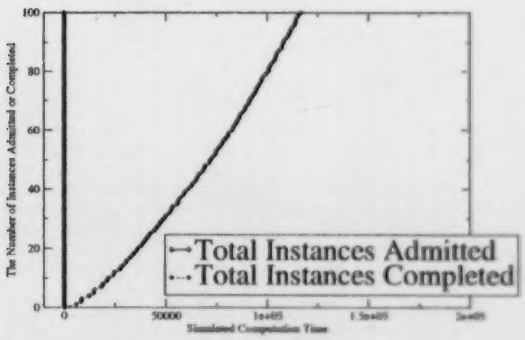
Again, for instances of all the benchmark workflows, the job service times and file sizes are assumed to be over-estimated and uniformly distributed on [500, 1000] time units and [1, 10] storage units, respectively. In the experiments we also assume that all of the workflow instances arrive at the same time.


The Number of Instances Admitted or Completed
Simulated Computation Time
Total Instances Admitted
Total Instances Completed

(a) DAR

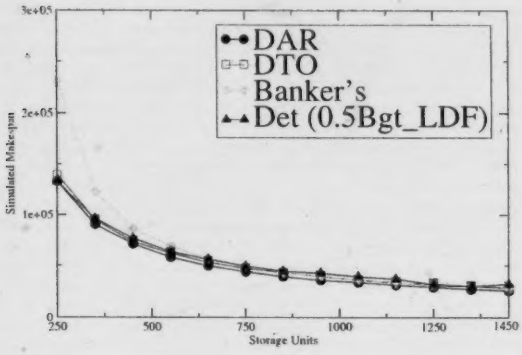

The Number of Instances Admitted or Completed
Simulated Computation Time
Total Instances Admitted
Total Instances Completed

(b) DTO

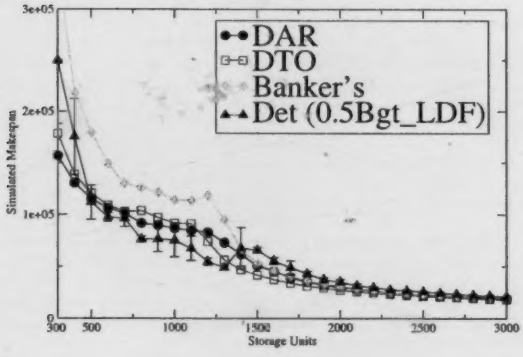

The Number of Instances Admitted or Completed
Simulated Computation Time
Total Instances Admitted
Total Instances Completed

(c) Banker's

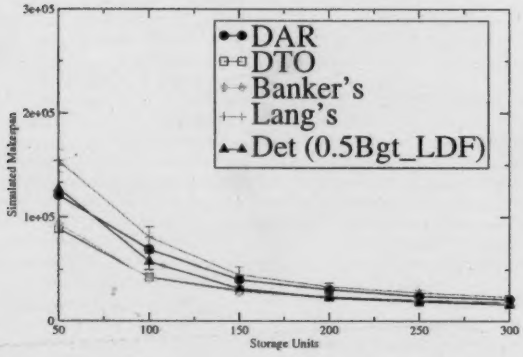
Figure 4.32. Execution Traces: The total number of workflow instances are admitted and completed by the compared algorithms as the computations proceed. The examined workflow is the Fork&Join$^+$ (3×32) with a multiple-reader access pattern, and the storage budget is 1400 units.



(a) Fork&Join (3 × 8)

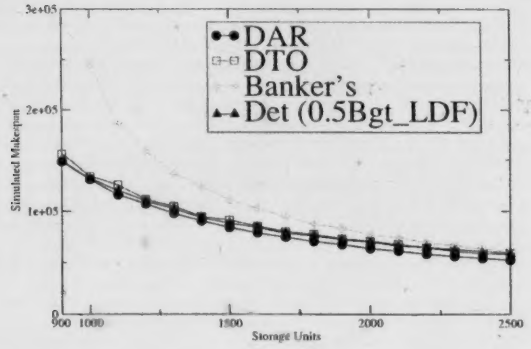


(b) Lattice (4 × 6)

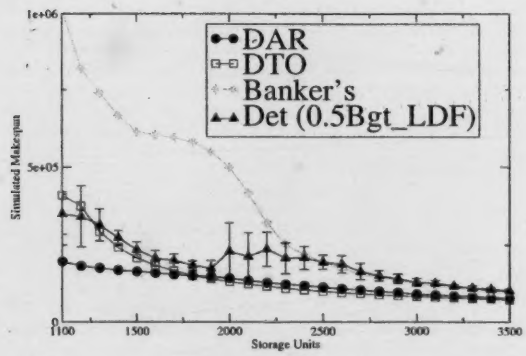


(c) Pipeline (5-stage)

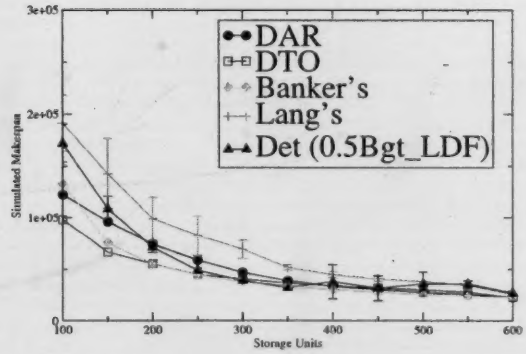
Figure 4.33. Makespan Comparisons (Small Workflow Size): The deadlock avoidance algorithms and detection algorithm Det(0.5Bgt_LDF) are compared when the workflow shape is changed. Each file has only one reader.



(a) Fork&Join (3 × 32)



(b) Lattice (8 × 12)



(c) Pipeline (10-stage)

Figure 4.34. Makespan Comparisons (Large Workflow Size): The deadlock avoidance algorithms and detection algorithm Det(0.5Bgt_LDF) are compared when the workflow shape is changed. Each file in the workloads has only one reader.

	Lattice (4×6) (2400 jobs)				Lattice (8×12) (9600 jobs)			
Storage Budget (unit)	300	400	500	600	1200	1600	2000	2400
# of deadlocks	7.67	2	2	1.67	1.9	1	0.7	0.1
# of rollbacked jobs	48.25	50.33	64.67	60.67	138.5	92.3	101.9	17.8
% of rollbacked jobs	2.01%	2.1%	2.69%	2.53%	1.44%	0.96%	1.06%	0.12%

Table 4.8. Comparisons of the Overhead of Deadlock Recovery between Lattice ($4_{\mathrm{e}} \times 6$) and Lattice (8×12): Note that the storage budget given to the Lattice (8×12) is 4 times as much as the budget given to the Lattice (4×6).

Figures 4.33 and 4.34 show how our proposed deadlock avoidance algorithms are *on average* better than the deadlock detection algorithm. More specifically, Figures 4.33(a) and 4.34(a) show that for the Fork&Join (single-reader situation) the performance of the proposed avoidance algorithms and detection algorithm is very close, regardless of the workflow sizes. This primarily results from the large fan-out factor of the workflow, which prevents an overly large number of instances from being admitted and thus minimizes the storage contention (i.e., the number of deadlocks and hence, the overhead of deadlock recovery).

In contrast, for the Lattice workload (see Figures 4.33(b) and 4.34(b)), the relative performance between the proposed avoidance algorithms and detection algorithm changes as the workflow size increases. When the workflow size is small (i.e. the Lattice (4×6)), the detection algorithm is not always inferior to the avoidance algorithms. Rather, it exhibits the best performance in some storage ranges. The reason behind this is that when the storage budget is *moderately* small (not too small, i.e., not less than 500 units in Figure 4.33(b)), the deadlocks generally occur in the early stage of the computation. As a result, the number of completed jobs in each rollbacked instance is small, and the recovery overhead is thus low. More importantly, the recovery process reallocates the storage to some deadlocked instances first instead of to the re-submitted victim instances, which will minimize the inactive storage and reduce the makespan. This demonstrates that for the detection algorithm, the recovery process is not always detrimental to performance. Rather, if the recovery overhead is not high, the storage reallocation may improve the overall performance.

However, with increasing Lattice size, the relative performance of the detection algorithm to DAR and DTO is degraded. At first, we intuitively attributed this result to the overhead of the deadlock recovery. However, based on our observations, the recovery overhead is unexpectedly non-increasing as the Lattice size increases. We can observe this from Table 4.8, where the number of deadlocks and the number and percentage of the total rollbacked jobs (i.e., recovery overhead) for both the Lattice(4×6) and Lattice (8×12) are listed for comparison. We found the reason is that as

	Pipeline (5-stage) (500 jobs)				Pipeline (10-stage) (1000 jobs)			
Storage Budget (unit)	50	100	150	200	100	200	300	400
# of deadlocks	8.8	1.7	1	1	8.8	2.2	1	0.8
# of rollbacked jobs	78.3	23.9	14.6	19.5	145.8	23.4	28	29.6
% of rollbacked jobs	15.7%	4.78%	2.92%	3.9%	14.58%	2.34%	2.8%	2.96%

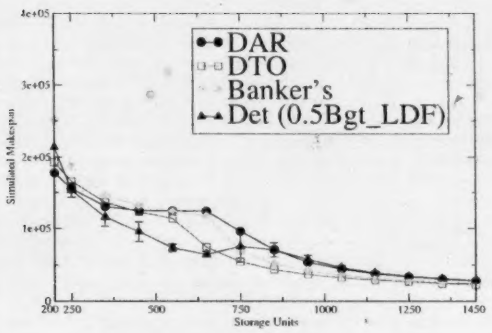
Table 4.9. Comparisons of the Overhead of Deadlock Recovery between Pipeline (5-stage) and Pipeline (10-stage): Note that the storage budget given to the Pipeline (10-stage) is twice as much as the budget given to the Pipeline (5-stage).

the Lattice size increases, the high level of data dependencies inside the Lattice workflow reduces the storage contention and thus minimizes the possibilities of deadlocks.

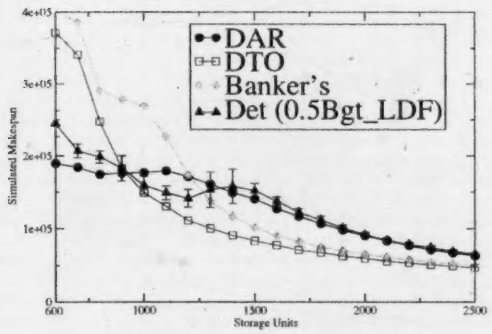
In fact, the major reason for the poor relative performance of the detection algorithm is that the performance of DAR and DTO is relatively improved as the Lattice size increases. More specifically, although the given storage budget is proportional to the workflow size in our experiments, the maximum claims computed by DAR and DTO increase relatively faster than the increased storage budget as the Lattice size becomes large (again, the maximum claim depends on the file sizes rather than the jobs in the workflow). The large value of the maximum claim can prevent the Lattice instances from being overly admitted, as discussed earlier. As a result, the performance of both DAR and DTO is relatively improved, compared to that for the small Lattice size, especially for DAR when storage is highly constrained. Therefore, in most cases, both DAR and DTO outperform the deadlock detection algorithm by up to 49.7%.

Figures 4.33(c) and 4.34(c) show how the relative performance between the detection algorithm and avoidance algorithms is largely unchanged for the Pipeline workloads when the number of stages increases from 5 to 10. For both Pipeline workflow sizes, when the storage is highly constrained (less than 50 and 200 units for the 5-stage and 10-stage Pipelines respectively), both DAR and DTO are better than the detection algorithm. However, as the available storage increases, the detection algorithm gradually outperforms the DAR algorithm, but in most cases the detection algorithm is not as good as the DTO algorithm.

We attribute these phenomena to the aggressive instance admission and high overhead of deadlock recovery in the detection algorithm. We can observe the evidence supporting this conclusion from Table 4.9, where the number of deadlocks and the number and percentage of the total rollbacked jobs (i.e., recovery overhead) for both Pipeline (5-stage) and Pipeline (10-stage) are listed for comparison. Compared to the Lattice workload (Table 4.8), the Pipeline workload has relatively high recovery overhead due to its low level of data dependencies inside the workflow as well as due



(a) Fork&Join$^+$ (3×8)



(b) Fork&Join$^+$ (3×32)

Figure 4.35. Makespan Comparisons: the deadlock avoidance algorithms and detection algorithm Det(0.5Bgt_LDF) are compared for the Fork&Join$^+$ (3×8) and Fork&Join$^+$ (3×32) with a multiple-reader access pattern.

to high storage contentions. However, as the storage increases, the number of deadlocks and total rollbacked jobs decrease (Table 4.9), and thus the performance of the detection algorithm improves rapidly, even over both DAR and DTO.

As the storage budget continues to increase, the overhead of the deadlock recovery may increase since approximately 50% of the storage budget needs to be released after a deadlock has been detected, which indicates that more admitted instances and jobs need to be rollbacked during the recovery. This is evidenced by the data shown in Table 4.9. For example, when the storage budget increases from 200 to 300 units for the 10-stage Pipeline workload, the number of rollbacked jobs increases from 23.4 to 28. Although the difference is not very large, the rollbacked jobs might

otherwise hold storage for a long time before they are rollbacked, adversely affecting the overall computation performance.

The same explanation can also be applied to similar observations for the Lattice workload (Table 4.8). For example, the makespan of the detection algorithm for the Lattice (8×12) is elongated when the storage budget increases from 1900 to 2000 units.

In addition, we compared the algorithms for the Fork&Join with a multiple-reader access pattern. To simplify the presentation, we will use Fork&Join$^+$ (3×8) to denote the Fork&Join (3×8) workflow with a multiple-reader access pattern in the following description (including Chapter 4.5.7). Figure 4.35 shows how the relative performance between the algorithms is changed as the workflow size is varied.

Figure 4.35(a) shows that in the case of small workflow size, the detection algorithm exhibits the best performance when the storage budget is moderately low (not less than 250 units), but as the storage increases, the detection algorithm gradually becomes worse and is outperformed by DAR and DTO. Performance behavior such as that of the detection algorithm for the Fork&Join$^+$ (3×8) is quite similar to that for the Lattice (3×8). We can understand this by following the same reasoning as for the Lattice workload.

Figure 4.35(b) shows that as the workflow size increases (i.e., to Fork&Join$^+$(3×32)), the relative performance of the detection algorithm remains largely unchanged and the performance differences with the avoidance algorithms are enlarged.

Similar to the Fork&Join$^+$ (3×8), both DAR and DTO are better than the detection algorithm for the Fork&Join$^+$ (3×32) when the storage budget is very tight (less than 900 units in Figure 4.35(b)). However, unlike the Fork&Join$^+$ (3×8), due to the initial large value of the maximum claim, DAR can prevent the instances from being overly admitted and thus exhibits a relatively large performance advantage over DTO and the detection algorithm, which is consistent with our observations in the Lattice (8×12) (Chapter 4.5.5.3). The detection algorithm outperforms DTO significantly for the Fork&Join$^+$ (3×32) because of (1) the aggressiveness of DTO in admitting instances (i.e., degrading the performance of DTO) and (2) the low recovery overhead as well as the benefits of reallocating storage in the detection algorithm (thereby improving the performance of the detection algorithm).

In summary, the relative performance between the proposed avoidance algorithms and the detection algorithm depends on the shape and size of workflow. By and large, we reach the following conclusions:

1. Regardless of the workflow size, DAR slightly and consistently outperforms the detection al-

gorithm for the Fork&Join workload with a single-reader access pattern. However, for the workflow with a multiple-reader access pattern, depending on the given storage budget, neither DAR nor DTO is consistently better than the detection algorithm.

2. For the Lattice, the avoidance algorithms (especially, the DAR algorithm) exhibit significant performance advantages over the detection algorithm when the workflow size is large. Otherwise, the detection algorithm generally performs better, but not always.

3. Both DAR and DTO outperform the detection algorithm when the storage is highly constrained for the Pipeline workloads. The performance differences between them become pronounced when the workflow size is enlarged. However, as the storage increases, the performance of the detection algorithm improves rapidly, even over both DAR and DTO.

These conclusions demonstrate that our deadlock avoidance algorithms are generally desirable when the storage resources are highly constrained and the workflow size is relatively large.

4.5.7 Performance Benefits of Instance Admission Control

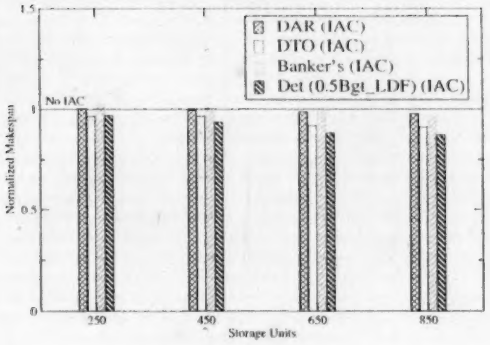
In this section we show how Instance Admission Control (IAC) benefits the performance of *all* the compared algorithms, including the deadlock detection algorithm, regardless of the workflow shapes, sizes and file access patterns.

Again, for instances of all the benchmark workflows, the job service times and file sizes are assumed to be over-estimated and uniformly distributed on [500, 1000] time units and [1, 10] storage units, respectively. In the experiments we also assumed that all of the workflow instances arrive at the same time.

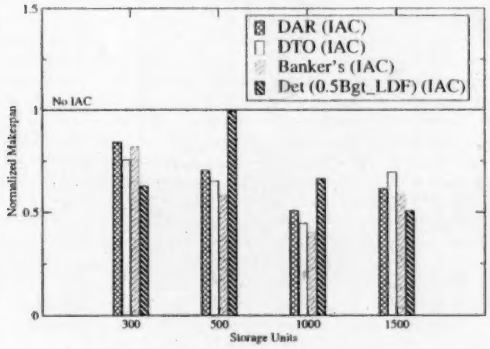
Our simulation results are shown in Figures 4.36 and 4.37, where the normalized makespans of the compared algorithms for the different workflow sizes are compared, given IAC or not. The normalized makespan is computed as $Makespan_{IAC}/Makespan_{NoIAC}$, where $Makespan_{IAC}$ and $Makespan_{NoIAC}$ are the respective makespans when IAC is present and when it is absent. From these figures we have two major observations:

1. The performance benefits of IAC are independent of workflow size, except for the deadlock detection algorithm on some data points for the Pipeline workloads.

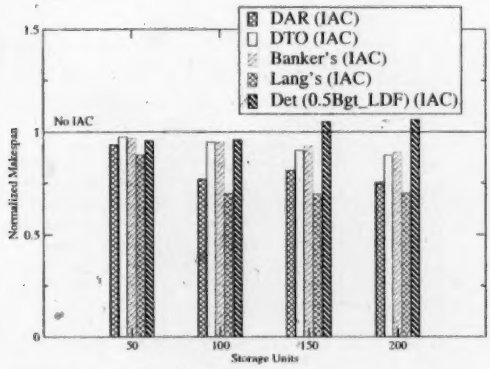
2. The performance benefits of IAC are highly sensitive to the workflow shape. Specifically, for the Fork&Join the benefits are marginal, but for the Lattice and Pipeline the benefits can be large.



(a) Fork&Join (3×8)

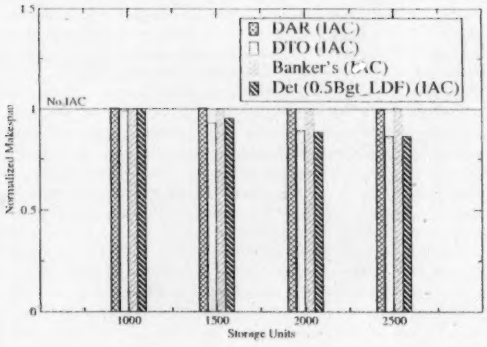


(b) Lattice (4×6)

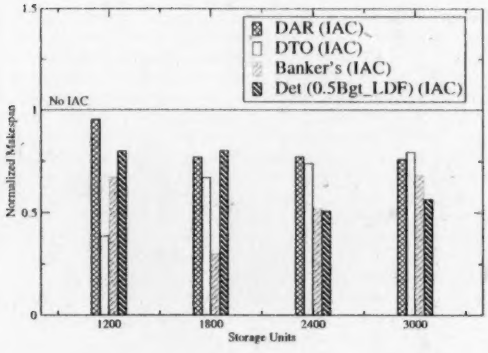


(c) Pipeline (5-stage)

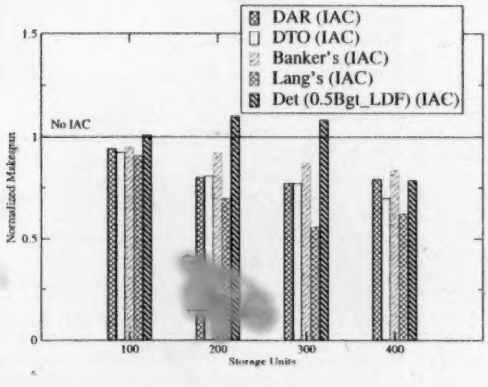
Figure 4.36. Performance Benefits of Instance Admission Control (IAC) Measured by the Normalized Makespan (Small Workflow Size)



(a) Fork&Join (3×32)



(b) Lattice (8×12)



(c) Pipeline (10-stage)

Figure 4.37. Performance Benefits of Instance Admission Control (IAC) Measured by the Normalized Makespan (Large Workflow Size)

We found that the major benefit of IAC is the admission control of the workflow instances (via the estimated number of concurrent instances) at the beginning of the computation so that a significant amount of storage can be reserved for the admitted instances. For the Fork&Join, regardless of the workflow size, due to the large fan-out factor, the difference between the estimated number of the concurrent instances and the actual number of the concurrent instances is small. Therefore, the performance benefits of IAC for the Fork&Join workload are marginal (see Figures 4.36(a) and 4.37(a)).

However, for the Lattice workload, regardless of the workflow size, significant performance benefits of IAC are exhibited in all the compared algorithms (see Figures 4.36(b) and 4.37(b)). For example, up to 69.7% (Lattice (8×32), 1800 storage units) performance improvement can be obtained by the banker's algorithm, up to 49.4% by DAR (Lattice (4×6), 1000 storage units) and up to 61.2% by DTO (Lattice (8×12), 1200 storage units) in the presence of IAC. We attribute these results to the large difference between the estimated instance concurrencies and the actual instance concurrencies. As a result, IAC exhibits more performance benefits for the Lattice workload.

Since there is no intra-instance concurrency in the Pipeline workloads, the estimated number of concurrent instances does not deviate much from the actual value. As a consequence, the performance benefits of IAC on the compared algorithms for the Pipeline are relatively small (see Figures 4.36(c) and 4.37(c)).

The detection algorithm, in general, demonstrates better performance in the presence of IAC because IAC not only minimizes the inactive storage but also has the side effect of reducing the impact of deadlock (the number of active instances is controlled). However, in some cases (see Figure 4.37(c) when the storage budget is 300), the performance of the detection algorithm in the presence of IAC is worse than that of the detection algorithm without IAC, which is different from the situations in DAR and DTO. This is not surprising since, as we know from the previous discussion, deadlock is not always detrimental to performance. On the other hand, the IAC based on the estimated number might not be always helpful to the performance improvements.

In addition, we investigate the benefits of IAC on all the compared algorithms for the Fork&Join$^+$ (3×8) and Fork&Join$^+$ (3×32). Figure 4.38 shows that, as the effects of the fan-out factor diminish, the performance benefits of IAC (measured by the normalized makespans) on all the compared algorithms become more pronounced. The results are different from those for the Fork&Join with a single-reader access pattern, where the performance improvements due to IAC are marginal. The reasons behind these results are not difficult to understand. By limiting the number of concurrent instances, IAC can dramatically improve the intra-instance concurrency of the Fork&Join workloads

	DAR		DTO		Banker's		Det(0.5Bgt,LDF)	
Storage	No IAC	IAC	No IAC	IAC	No IAC	IAC	No IAC	IAC
250	131517	131934	138139	133138	229379	234319	134049	**129732**
350	**89841.4**	90395.7	94735.6	92141.3	121654	122609	95811.2	90728.3
450	70164.5	**69808**	74150.5	71486.7	86165.2	86578.7	74837.5	69902
550	**57712.4**	58043	61578.6	58907.7	68581.2	68826.7	64209.3	57910.3
650	49724.1	49040	53423.4	48994.3	56791.6	55828.3	55148.3	**48545**
750	43902.7	42802	46591.3	42911.7	49418.2	47801.7	50021.8	**42608.3**
850	39192.7	38315.7	42058	38319	43531.4	41971.7	43374.3	**37723**
950	35790	34732.3	39403.9	34485.7	39502.9	37502	41597.9	**34163**
1050	33462	31749	36228.2	31755.3	36224.7	34367.3	38321.2	**31485.3**
1150	31145.5	28931	33762.4	29074.3	33495.2	30515.3	36490.6	**28919**
1250	29142.5	26965	31545.6	27054	30905.9	28633.3	34253.5	**26902**
1350	27511.7	25319.3	30211	**25158.7**	29362.8	26534.3	32133.9	25364
1450	26174.2	24119.7	28702.1	**23845.7**	27806.7	25123.7	32136.5	23944.3

Table 4.10. Performance Benefits of Instance Admission Control (IAC) Measured by Makespan: The algorithms are compared (measured in time units) for the Fork&Join (3×8) workload when IAC is present and when it is absent. Storage budget is varied from 250 to 1450 storage units. The lowest makespan in each row is boldfaced.

because of the Pipeline structure inside the workflow and diminished effects of the fan-out factor.

Finally, in order to compare the relative performance between the compared algorithms when IAC is present and when it is absent, we show more data points in Tables 4.10 through 4.13 for the small workflows (i.e., Fork&Join (3×8), Lattice (4×6), Pipeline (5-stage) and Fork&Join$^+$ (3×8)); and in Tables 4.14 through 4.17 for the large workflows (i.e., Fork&Join (3×32), Lattice (8×12), Pipeline (10-stage) and Fork&Join$^+$ (3×32)). In each table the lowest makespan in each row is boldfaced, which indicates which algorithm, in combination with IAC or not, can demonstrate the best performance.

From these tables we have the following observations:

1. Regardless of the workflow size, the detection algorithm combined with IAC in most cases gives the best performance for Fork&Join. However, for Fork&Join$^+$, in most cases the best result is achieved by DAR combined with IAC.

2. DTO in combination with IAC is consistently the best for the Lattice workloads, regardless of the workflow size.

3. When IAC is present, DTO and banker's are competitive in showing the best performance for the Pipeline (5-stage). However, for the Pipeline (10-stage), DTO combined with IAC is clearly the performance leader.

These observations demonstrate that, in general, it is more valuable to combine IAC with DAR and DTO than with the reference algorithms to achieve better performance. We think the reason for

	DAR		DTO		Banker's		Det(0.5Bgt_LDF)	
Storage	No IAC	IAC	No IAC	IAC	No IAC	IAC	No IAC	IAC
400	130244	105403	138664	**98643.8**	218508	151242	175311	153770
600	106076	72625.9	108776	**65511.4**	149053	82590.6	97400	91166.1
800	91639.7	54745	103539	**49903.7**	125965	58595.6	76074	65129
1000	86653	43861.8	91597.7	**40692.1**	114097	45872.7	74732	49518.2
1200	82756.3	37315.6	73836.7	**34684.5**	118025	38378.5	54626	40468.3
1400	61705	32272.3	46821.7	**30906.1**	65915	32956.1	67590	35497.4
1600	45072.3	29330.6	37645	**27738.4**	45722	29342.3	55304.3	32011.1
1800	35531	26218.5	31455.3	**25166**	36355.3	26847.4	42964.3	28621.1
2000	30396.3	24237.7	27485	**22757.1**	30850.3	24312.7	35409.3	25853.2

Table 4.11. Performance Benefits of Instance Admission Control (IAC) Measured by Makespan: The algorithms are compared (measured in time units) for the Lattice (4×6) workload when IAC is present and when it is absent. Storage budget is varied from 400 to 2000 storage units. The lowest makespan in each row is boldfaced

	DAR		DTO		Banker's		Lang's		Det(0.5Bgt_LDF)	
Storage	No IAC	IAC	No IAC	IAC	No IAC	IAC	No IAC	IAC	No IAC	IAC
50	120503	112850	88661.4	**86513.7**	93133.3	90346	154004	136907	126967	121578
100	69052.7	52939.1	42452.6	40374.1	41696.8	**39698.5**	80590.2	56266.2	57655	55427.9
150	40066	32457	29613.7	26935.4	28415.6	**26535.5**	44952.5	31335.8	31416.1	32968.3
200	30561	22913.3	23426.6	20743.3	22604.7	**20357.2**	33242.9	23332	22470.7	23802.2
250	24154.5	18311.7	20118.2	16844	19459.3	**16806.1**	27342	18287.9	18896.4	18349.1
300	20984.3	15439.6	17883.4	**14531.9**	17410.7	14547.4	23416.5	15540.4	15938.7	15570.2

Table 4.12. Performance Benefits of Instance Admission Control (IAC) Measured by Makespan: The algorithms are compared (measured in time units) for the Pipeline (5-stage) workload when IAC is present and when it is absent. Storage budget is varied from 50 to 300 storage units. The lowest makespan in each row is boldfaced.

this is that the reference algorithms are either too aggressive (e.g., the detection algorithm) or too conservative (e.g., the banker's algorithm) in granting the job resource requests. In the former case, when the number of the active instances is less than the estimated value during the computation, new instances are admitted more easily, potentially increasing the storage competition and the amount of inactive storage, whereas in the latter case the reserved resources due to the presence of IAC might not be efficiently used. In contrast, both DAR and DTO, by exploiting the dataflow information, can make efficient use of the storage resources and thus are more beneficial than the reference algorithms in combination with IAC.

4.6 Concluding Remarks

This chapter studied the value of dataflow information to a deadlock problem in workflow-based computing when storage resources are constrained. To this end, we presented two dataflow-based deadlock avoidance algorithms (i.e., DAR and DTO) based on the well-known banker's algorithm.

	DAR		DTO		Banker's		Det(0.5Bgt_LDF)	
Storage	No IAC	IAC	No IAC	IAC	No IAC	IAC	No IAC	IAC
250	155600	**102528**	165829	116919	186795	134431	153235	103991
350	130579	**73314.9**	136258	78687.3	146153	85515.9	115678	75961.3
450	124860	**57538.5**	123098	60128.2	132095	63359.6	95367.1	59488.9
550	124536	**47316.8**	113730	49576.8	124346	51252.7	73315.2	48922.8
650	123896	**41019.4**	74467.4	42127.3	117350	43623.1	64464	43356.1
750	95688.5	**36253.7**	54318.9	36545.1	69178.6	37474.8	75253	37826.5
850	70996.6	**32281.2**	43698.4	32874.3	51492.2	33613.1	70548.1	33771.1
950	53772.6	**29485.1**	36908.8	29516.3	42404.3	30431.7	57641.1	31004.9
1050	44756.6	27147.5	32242.2	**27061.9**	36005.5	27661.1	46265.9	28399.2
1150	37982.3	25155.4	28814.8	**25119**	31621.5	25511.3	39300.6	26639.7
1250	33536.5	23514.5	26239.3	**23327.8**	28494	23787.2	34291.1	24903.1
1350	30048.3	22135.6	24063.4	**22124.6**	25692.2	22396.2	30707	23544.2
1450	27387	20813.2	22475.4	**20717.3**	23722.5	21259.3	27858.7	22324.9

Table 4.13. Performance Benefits of Instance Admission Control (IAC) Measured by Makespan: The algorithms are compared (measured in time units) for the Fork&Join$^+$ (3×8) workload (multiple readers) when IAC is present and when it is absent. Storage budget is varied from 250 to 1450 storage units. The lowest makespan in each row is boldfaced.

	DAR		DTO		Banker's		Det(0.5Bgt_LDF)	
Storage	No IAC	IAC	No IAC	IAC	No IAC	IAC	No IAC	IAC
1000	131023	131310	134044	132185	245419	241950	131772	**129885**
1200	107134	106357	111314	107567	158998	157842	109793	**106115**
1400	90634.2	90234.3	94784.9	91713.5	124120	123949	93786.2	**90218**
1600	79110.4	79045.8	85391.7	79782.7	102918	102530	84427.1	**78959**
1800	70509.6	70186.9	77752.6	71117	87649.1	87677.4	76395.7	**70183.1**
2000	63757.4	**63573.3**	70910	64142.1	77720.2	77774.3	70122.7	63657.3
2200	58438.6	**58265.9**	65002.3	58605	69750.2	69825.8	64971.3	58289.7
2400	54253.9	54162.1	61190.2	54044.8	63772	63434	59069	**53698.1**

Table 4.14. Performance Benefits of Instance Admission Control (IAC) Measured by Makespan: The algorithms are compared (measured in time units) for the Fork&Join (3×32) workload when IAC is present and when it is absent. Storage budget is varied from 1000 to 2400 storage units. The lowest makespan in each row is boldfaced.

The essence of these algorithms is to make important distinction between active and inactive resources and attempt to maximize the active resource utilization for performance while avoiding deadlock.

Through simulation-based studies, we show how dataflow information allows our DAR and DTO to have lower makespans than the control-flow-based banker's algorithm, Lang's algorithm and the deadlock detection algorithm for a variety of workflow shapes, sizes, and other parameters.

	DAR		DTO		Barker's		Lazy's		Eager (Sig. LDF)	
Storage	No IAC	IAC	No IAC	IAC	No IAC	IAC	No IAC	IAC	No IAC	IAC
1200	181700	170328	375807	**150044**	830535	553011	542360	276200		
1400	168563	141376	242077	**128602**	666974	315651	273393	206986		
1600	159435	128902	183754	**113618**	604108	231969	206670	168375		
1800	151154	112890	151374	**101434**	581179	182276	164012	142270		
2000	143084	106324	131850	**93393.6**	498376	154059	139940	128370		
2200	128633	96364.3	116970	**86155.1**	322673	130891	126431	116630		
2400	119226	88926	105149	**78719**	219772	116226	109678	101209		
2600	109070	82907.3	96411.7	**74661.9**	175634	103660	98726	91488.3		
2800	100355	76944.8	89562.1	**69286.7**	147842	94725.1	90710	85089.5		
3000	92046.7	71025.1	83680.8	**66449.3**	127288	86703.3	83855	78163		
3200	86875.5	68182	78949.4	**62292.9**	114022	80437	78171	73738		
3400	80708.7	64381.5	75133.3	**58660.3**	101947	75215.9	73955	69599.6		

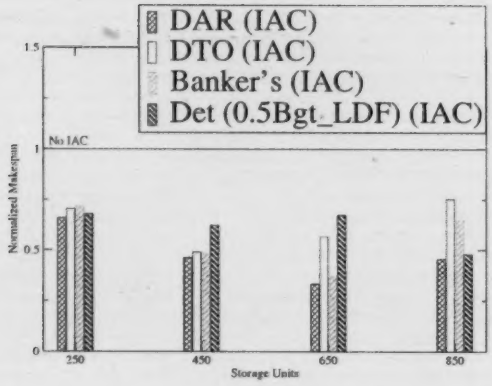
Table 4.15. Performance Benefits of Instance Admission Control (IAC) Measured by Makespan: The algorithms are compared (measured in time units) for the Lattice (8×12) workload when IAC is present and when it is absent. Storage budget is varied from 1200 to 3400 storage units. The lowest makespan in each row is boldfaced.

	DAR		DTO		Barker's		Lazy's		Eager (Sig. LDF)	
Storage	No IAC	IAC	No IAC	IAC	No IAC	IAC	No IAC	IAC	No IAC	IAC
100	122096	114920	97967.3	**90564.6**	132994	126164	194251	173295	171910	171207
200	74347.3	59413	55442.7	**44784.4**	55302.3	50779.2	90990.7	68936.2	71307.3	70330.8
300	46954.7	36141.6	40532.7	**31192.2**	38092	33233.4	69593	50909	50907.3	47349.2
400	33397.7	26402.2	33987.7	**23697**	30604.3	25532.6	45809	38057.6	37998.7	35838.6
500	27774.3	21588.6	29977	**21092.8**	26244	21752.8	38089	32637.2	35860	33039.6
600	22909	18574.6	22732.3	**16664.4**	22853	18185.2	25860	19227.6	26926	20302.4

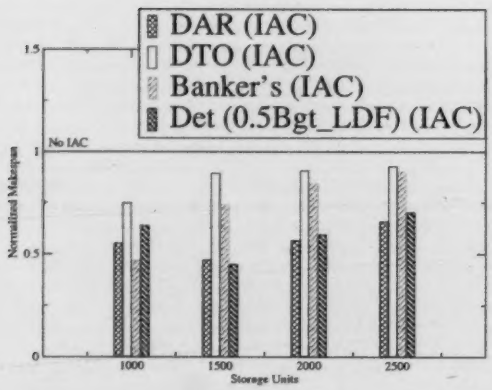
Table 4.16. Performance Benefits of Instance Admission Control (IAC) Measured by Makespan: The algorithms are compared (measured in time units) for the Pipeline (10-stage) workload when IAC is present and when it is absent. Storage budget is varied from 100 to 600 storage units. The lowest makespan in each row is boldfaced.

	DAR		DTO		Barker's		Eager (Sig. LDF)	
Storage	No IAC	IAC	No IAC	IAC	No IAC	IAC	No IAC	IAC
1000	177012	**97265.8**	149178	113000	2091129	128468	1598119	1010225		
1200	171605	**81741.2**	111102	93860.7	173712	101252	1401579	870814.6		
1400	149194	**79220.4**	90961.9	77850.1	116987	82564.9	1180460	740174.8		
1600	127126	**61616.6**	77693.4	67984.6	90801.6	70995	1177889	657362.6		
1800	106901	**56328.7**	67676.2	61470.5	79556	62562.9	1010849	608992.7		
2000	90396.1	**51138.6**	59740.4	53031.5	65402.9	56420.6	910277.9	549785.6		
2200	77527.6	**47057.1**	53550.7	50494.3	57997.2	51259.3	786514.6	507005.7		
2400	67196.4	**43623.4**	49075.1	47226.7	52210.5	47015.2	689776.9	466614.2		

Table 4.17. Makespan Comparison: The algorithms are compared (measured in time units) for the Fork&Join$^+$ (1×32) workload (multiple readers) when IAC is present and when it is absent. Storage budget is varied from 1000 to 2400 storage units. The lowest makespan in each row is boldfaced.



(a) Fork&Join$^+$ (3×8)



(b) Fork&Join$^+$ (3×32)

Figure 4.38. Performance Benefits of Instance Admission Control (IAC) Measured by the Normalized Makespan: DAR, DTO and the banker's algorithm are compared in the absence and presence of IAC for the benchmark workloads of Fork&Join$^+$ (3×8) and Fork&Join$^+$ (3×32) with a multiple-reader access pattern.

Chapter 5

WaFS Prototype

The main point of this research is that dataflow information is useful when scheduling scientific computations. Specifically, a variety of scheduling algorithms have been proposed and evaluated through a simulation-based, parameter space study (Chapters 3 and 4). However, since the scheduling benefits are predicated on having the dataflow information in the first place, we address the issue of how to obtain the necessary information for a workflow-aware file system.

In this chapter, we describe the design and implementation of a WaFS prototype. As a proof-of-concept, the WaFS prototype provides evidence that such a system can be built with low (e.g., 12% or less) overheads and that the dataflow, job service time (JST), and file size information can be gathered transparently. Finally, the information gathered by the WaFS prototype is used as the baseline values for a new simulated workload, with qualitatively similar (but, as expected, quantitatively different) results to previous simulations, thus validating some of the ideas from this research.

5.1 Design Options

We have developed a prototype as a proof-of-concept, called *WaFS* (short for *Workflow-aware File System*), to efficiently collect and exploit file-based dataflow information on a per-instance basis. The dataflow information is collected by tracking the system calls `open()` and `close()` to determine producer-consumer relationships. Thus, a key design problem in WaFS is how to intercept system calls related to file accesses. This can be done either at the kernel level or at the user-level.

One of the most natural kernel-level approaches would be to directly modify the system call routines that need to be monitored. However, this approach requires modifying the kernel and hence rebuilding the kernel. Another relatively simple kernel-level approach is to write a *Loadable Kernel*

Module (LKM) that can replace an entry in the *system call table* of the kernel with a function of our own. Our function would then be invoked when a system call is made, instead of the kernel code implementing the invoked system call (e.g. `open()`). Although this approach does not require the modification and rebuild of the kernel, the major problem is that we have to know the address of the system call table, which is no longer given after Linux Kernel 2.4. Since we are only concerned with the system calls that are related to file system accesses, we have another option for tracking these system calls by installing an extensible file system via the *Virtual File System* (VFS) (e.g., Scruf [68]).

Kernel-level approaches are transparent to the user applications and execution environments and generally demonstrate good performance in observing all file access operations made by a given process. However, these approaches suffer from a major drawback in that the load and unload of the modules require super-user privileges, making them difficult to deploy in practice. Therefore, the most user-friendly approach would be a method acting at just the user level.

Our prototype consists of three major components: *Batch Scheduler*, *Monitor* and *Versioned Namespace Manager* (VNM). As previously discussed, the Scheduler schedules the submitted jobs to computational hosts, and the VNM is responsible for detecting and managing the dataflow information associated with the workload. The Monitor mediates between Scheduler and VNM to intercept the system calls of the running jobs and communicate with VNM to allow VNM to accomplish its tasks. All these components are running at the user-level. Because of this, the architecture enjoys the following advantages.

1. No requirement to change the underlying file system.
2. No requirement to modify the standard shared libraries.
3. No requirements of the source code of the applications.

5.2 The Batch Scheduler and Versioned Namespace Manager

Note that our simulated batch scheduler is not as sophisticated as a production batch scheduler. Modifying PBS [47], LSF [108], or SGE [90] to be WaFS-compatible would be a desirable but non-trivial task. Furthermore, as justified earlier, a simulated system (including our simple batch scheduler) is what makes it realistic to explore the already large parameter space of our policies. Conversely, making our simulated batch scheduler as sophisticated as, say, PBS would add many more dimensions to the parameter space. Therefore, we focus our prototyping efforts on the least-

understood component of the overall system: How can we monitor jobs to transparently gather dataflow (and other) information (Chapter 5.3)?

We designed the batch scheduler and VNM based on a client/server architecture. The batch scheduler is *simulated* as a batch queuing and workload management system that acts as a client that communicates with the VNM (the server) to schedule and manage the user submitted workloads among a set of networked computational hosts.

The VNM is built on top of an existing file system to capture the dataflow information on a per-instance basis (i.e, multi-version namespace) and provide service to the client. To achieve this goal, an *On-line Data Dependency Solver* is implemented to construct dataflow graphs in WaFS by resolving the data dependencies as the computation proceeds. In addition, a query utility is provided to serve the remote requests (i.e., for dataflow information) from the batch scheduler.

The centralized VNM introduces a performance bottleneck, but we believe that the performance loss due to the centralized VNM accesses is insignificant compared to the job computation time.

5.3 The Monitor

There exist a variety of techniques that can be used in the Monitor to intercept the system calls at the user-level. These include wrapping the calls in the application source code, instrumenting the C library and overriding the C library. Each technique has its own drawbacks: source code required, language-dependent or not guaranteed to work with all processes.

The technique we used is the process-tracing mechanism called *ptrace* that was originally provided in the Linux kernel and most other Unix-like operating systems to facilitate debugging programs. Ptrace allows one process to examine and change the behavior of another process. This functionality is used by the debugging utility *strace* in Linux to monitor the system calls used by a program and all the signals it receives.

Using a ptrace-based monitoring system has all of the benefits (except the performance) of the kernel-level approaches and the previously discussed user-level approaches, yet it overcomes their respective drawbacks. For example, it is able to intercept all file access system calls made by any given process without the modification of the execution environments and the legacy applications. On the other hand, compared with the aforementioned user-level approaches, it also overcomes the difficulties in preserving and sharing information between monitored processes.[1]

The major disadvantage of the ptrace-based monitoring system is the performance degradation,

[1] The file system runs in the separate address space of each process; sharing data among them becomes more difficult.

especially for those system-call-intensive applications, as each system call may incur multiple context switches. However, we will provide evidence (Chapter 5.4.2) that the overheads are less than 12% for short jobs and less than 0.4% for the longer jobs of many HPC workloads.

5.3.1 Handling open() and close()

To collect dataflow information on a per-instance basis, it is critical for the Monitor to track both `open()` and `close()` system calls. Roughly speaking, intercepting the `open()` system call is primarily useful to track the dataflow dependency between the jobs, whereas intercepting the `close()` system call can assist in the construction of the versioned namespace for each workflow instance.

When entering the `open()` system call, the Monitor first checks whether the operation flag is `O_RDONLY` or `O_RDWR`. If it is, the Monitor obtains the file pathname[2] and sees if the file already exists. The file will be read directly if the answer is "Yes". Otherwise, the Monitor will contact the VNM to obtain the *correct* version of the file and its new location. However, if the checked file is not found in VNM, the Monitor will simply report "file not found in VNM" and continue the system call. Otherwise, the Monitor rewrites the file pathname argument of `open()` and continues the system call.

Before leaving `open()`, the Monitor creates an entry in a *file table* for each output file, recording its file descriptor and corresponding file name and operation flag.

The Monitor catches `close()` only before it leaves the system call. Specifically, the Monitor first uses the file descriptor to search the file table and get the corresponding file access information. If the file is created for writing, the Monitor first obtains a version number and a new path location from the VNM and then constructs a new pathname for the file and records it in the VNM. Finally, the Monitor moves the newly created file (with a new version number) to the new path location.

In addition to `open()` and `close()`, in order to handle the complexity of real applications, we need to pay attention to *all* the system calls that take a file pathname as an argument. Basically, the Monitor handles the file pathname by following a similar procedure.

5.3.2 Manipulating the File Pathname

To correctly collect the dataflow information, file pathname manipulation is critical. In our implementation we manipulate the file pathnames in two main cases: *linking* and *rewriting*.

[2] Here we do not discuss the filename canonicalization and the procedure of filtering out the non-relevant filenames.

Both *hard linking* and *soft linking* can affect the correctness of the detected dataflow graph if we do not consider them. One of the typical algorithms used to address the linking problem is to canonicalize paths by replacing such elements (along the path) as `'/'`, `'.'`, `'..'` and symbolic links with their absolute path. The GNU C library functions `realpath()` and `canonicalize_file_name()` can accomplish this functionality and get the real name of a file. However, both functions resolve relative paths from the working directory of the current running process, and hence cannot completely fulfill our requirements. Our solution to this problem is to simply change the working directory of the Monitor to that of the process being monitored before invoking the function of either `realpath()` or `canonicalize_file_name()` and then change back after finishing the function.

Rewriting the file pathname in the `open()` system call is always required in our implementation since the input file might be versioned and relocated somewhere. Unfortunately, it is not always possible to update the file pathname in place because the new pathname may be longer than the existing pathname, and the memory segment of the existing pathname may be read only. To address this issue, we adopted one of the typical solutions [2] that writes the new file pathname in a free portion of the monitored process's address space and redirects the system call argument (i.e., `EBX` register in Linux on x86) to point to the new pathname string.

5.3.3 Tracing Process Family

Tracing a process' family is another requirement for detecting and constructing the correct dataflow graphs for those real application workflows whose constituent jobs may involve multiple processes. For example, the jobs may invoke `system()` and/or `popen()` functions to execute other programs or invoke the system calls `fork()` or `clone()` directly to create multiple processes. However, the ptrace mechanism cannot automatically intercept the system calls that are made in the resulting child processes of the monitored process. To address this issue, one typical solution is to intercept the `fork()` system call made in the monitored process, obtain its child process id before `fork()` returns and then attach the child process to the monitoring process using `PTRACE_ATTACH` primitives. Although this solution is simple, it has a race condition since the newly attached process (i.e., the child process of the original monitored process) may start making system calls before its parent (i.e., the original monitored process) leaves the `fork()` system call.

A natural solution to the race condition is to not allow the child process to return from the `fork()` system call until the parent returns. This solution can be simply achieved by rewriting the intercepted `fork()` system call into a `clone()` system call with the `CLONE_PTRACE` flag in

our Linux platform.

5.4 Proof-of-Concept and Results

The prototype was created mainly as a proof-of-concept: to show it is possible, given certain assumptions about applications and workflows (e.g., static workflow shapes, all dataflows are via producer-consumer pattern), to automatically detect and collect the necessary dataflow information. In this section, we use GROMACS as an example. First, we show that the WaFS prototype can collect dataflow information from GROMACS. Second, we use the JST and file size information gathered by WaFS as the basis of a new, simple simulation study (Figure 5.5). The relative performance of our new policies (e.g., DAR, DTO) compared to existing policies (e.g., the banker's algorithm) are similar to the results in Chapter 4, but with significant differences in the absolute performance.

Ideally, the WaFS prototype should be directly integrated with a batch scheduler and Chapter 5.4.3 should be a real workload experiment instead of a simulation. However, building such a system is beyond the scope of this thesis.

Nonetheless, this section shows an end-to-end example of a real application (i.e., GROMACS), real dataflow, JST, and file size information gathered by the prototype, which is then used as baseline parameters for a simulation.

5.4.1 Dataflow Collection: A Running Example

To illustrate the capability of WaFS to automatically collect the dataflow information of a workflow, in this section we present a running example from *GROMACS* [42], i.e., the *Ribonulease S-peptide* (abbreviated as *S-peptide*) workflow. This example is interesting because of the following features:

- A detailed description of the steps to run the workflow and an example dataflow chart are available in the GROMACS document, which can be used to verify our results.
- The workflow has complicated file access patterns. For example, the constituent programs (i.e., jobs) can version their output files and call other programs through the *system()* function. Also, the programs can automatically add file extension names to the opened files (provided as parameters in the command line), depending on the command line options.

Roughly speaking, this workflow consists of five steps, which are implemented by seven programs (shown in parentheses):[3]

1. Generate a topology file (`pdb2gmx`),
2. Solvate the peptide (`editconf`, `genbox`),
3. Energy minimization (`grompp`, `mdrun`),
4. Molecular dynamics with position restraints (`grompp`, `mdrun`), and
5. Checking of the simulation results (`g_energy`).

To simulate the *S-peptide*, we need a starting structure (i.e., the initial input data). This can be taken from the protein data bank. Here we simply use the example stored in the file `speptide.pdb` in the GROMACS package. This file contains 146 atoms, classified into 19 groups.

We follow the semantics of *DAGMan* [18] to design our input script and show an example script of *S-peptide* workflow, called *gromacs.st*, in Figure 5.1 where three segments must be specified. The first includes the user name and the workflow name and location (i.e., the set of the constituent jobs in the workflow). For this particular instance the locations of the initial input data files and final output data files are specified by `Input` and `Output`, respectively. For example, the instance's inputs may be located in `Workflow/gromacs/init_input`, and the outputs in `Workflow/gromacs/final_output`.

The next segment is `.map` in which the jobs in the workflow are mapped one-to-one to integers, each integer representing a job and being used in the following segments.

The last segment *.dep* contains the control-flow description described by the user. For example, the child `Job 1` cannot start until its parent `Job 0` finishes. In our example, there are 8 jobs in the workflow which are organized as a pipeline (see Figure 5.2).

To submit a workload to our simulator, we also implement a command *qsub* to mimic submission in a real batch queuing system:

```
% qsub gromacs.st
```

qsub will parse the input script `gromacs.st` and translate its contents into a workflow instance that can be scheduled by the simulated batch scheduler. The dataflow dependencies are tracked by WaFS as the computation proceeds, and the computation ends up with the construction of the dataflow graph of the *S-peptide* workflow in VNM (Figure 5.3).

[3]Details at `http://www.gromacs.org/documentation/reference/online/speptide.html`.

```
# GROMACS script
# edited by Wang Yang
User: wang
Workflow: GROMACS
Location: Workflow/gromacs
Input:  Workflow/gromacs/init_input
Output:  Workflow/gromacs/final_output
.map
# jid   command arguments
# (1) Generate a topology file (.top) from the pdb-file (.pdb)
JOB 0   pdb2gmx -f speptide.pdb -p tmp/speptide.top -o tmp/speptide.gro
# (2) Solvate the peptide in a periodic box filled with water
JOB 1   editconf -f speptide -o -d 0.5
JOB 2   genbox -cp out -cs -p speptide -o tmp/b4em
# (3) Perform an energy minimization of the peptide in solvent
JOB 3   grompp -v -f em -c b4em -o tmp/em -p speptide
JOB 4   mdrun -v -s em -o tmp/em -c tmp/after_em -g tmp/emlog
# (4) Perform a short MD run with position restraints on the peptide
JOB 5   grompp -f pr -o tmp/pr -c after_em -r after_em -p speptide
JOB 6   mdrun -v -s pr -e tmp/pr -o tmp/pr -c tmp/after_pr -g tmp/prlog
# (5) Show results
JOB 7   g_energy -f pr -o tmp/out -w
.dep
# control-flow description
PARENT  0 CHILD 1
PARENT  1 CHILD 2
PARENT  2 CHILD 3
PARENT  3 CHILD 4
PARENT  4 CHILD 5
PARENT  5 CHILD 6
PARENT  6 CHILD 7
#end
```

Figure 5.1. An Example of a GROMACS Input Script: gromacs.st

5.4.2 Overhead of Prototype on Potential Applications

The overhead of WaFS on potential applications is expected to be low because there are relatively few calls to `open()` and `close()` in scientific workloads. This can be validated by examining the GROMACS benchmarking system *gmxbench* [42] that consists of four molecules published by the GROMACS group. The four molecules in the benchmark are *d.dppc*, *d.lzm*, *d.poly-ch2*, and *d.villin*, whose atom trajectories, in water, over a period of time, are simulated by GROMACS software. The *gmxbench* is known to be compute-intensive and is representative of a large class of simulation-based applications.

For our study, we compare the runtime of the computationally intensive `6.mdrun` program that actually performs the simulation. The configuration for this experiment is shown in Table 5.1. We

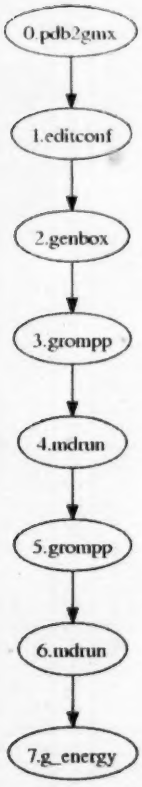


Figure 5.2. The Given CFG

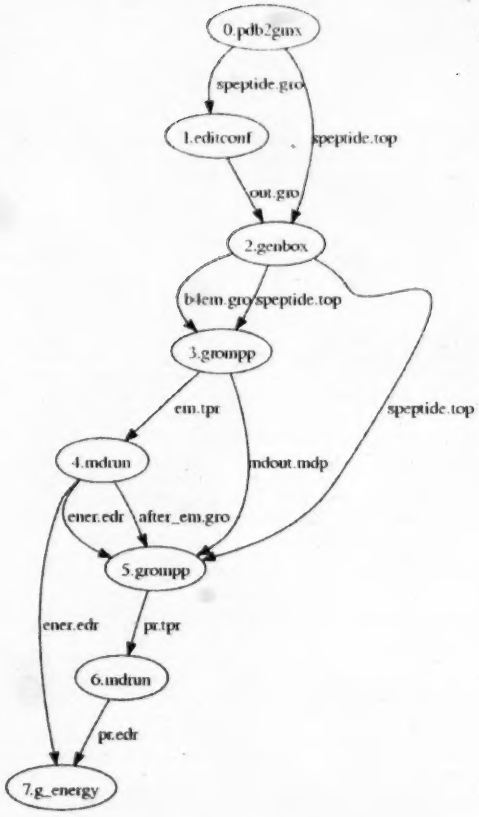


Figure 5.3. The Detected DFG

use two computers; one is assigned to the simulated batch scheduler and monitor, and the other is to the VNM. The monitor runs in each compute host to monitor the job execution and send the intercepted information to the remote VNM. The monitor and VNM constitute the WaFS prototype. The network between the Simulated Batch and VNM is 1 Gbit/s Ethernet.

The results of the GROMACS *gmxbench* are shown in Figure 5.4. Each data point is averaged over 5 runs. The bars labeled `Orig` are for the same runs, but without the overheads associated with WaFS. The overheads for WaFS are from 0.39% (*d.dppc*) to 11.74% (*d.villin*), depending on the computation involved in each simulated molecule. Although 11.74% overhead might be considered high in absolute terms, the low run time (97 seconds, Table 5.3) is not as typical as the longer runs times of the *d.dppc* example.

Component	CPU	Memory	Cache	OS
Simulated Batch (Monitor)	AMD Athlon 2.4GHz	1GB	512KB	Linux 2.4.29
VNM	AMD Athlon XP 2.2GHz	1GB	512KB	Linux 2.6.18

Table 5.1. Experimental Configuration for *gmxbench*

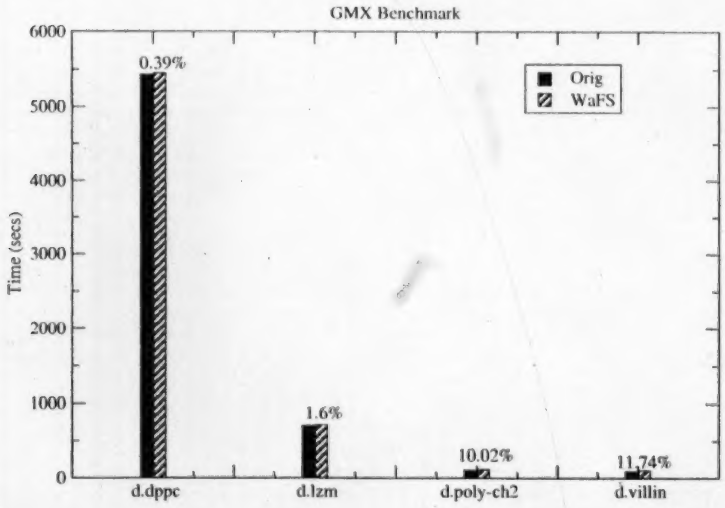


Figure 5.4. Performance of GROMACS *gmxbench* in absence and presence of WaFS. The performance overhead of WaFS is shown by the percentage above the bar.

5.4.3 Simulation Results for an GROMACS Workload

Of course, the purpose of WaFS is to gather information which can then be used by a WaFS-compatible batch scheduler. Unfortunately, as discussed earlier (Chapter 5.2), a WaFS-compatible scheduler does not exist and creating one is beyond the scope of the current research.

As an intermediate step, we can use the WaFS-measured information (e.g., dataflow, JST, file sizes) from real runs of GROMACS (Tables 5.2 and 5.3) to serve as the basis for a simulated workload (Figure 5.5). Note that the simulations in Chapters 3 and 4 are *not* based on these measured values because of the relative lack of variety (i.e., limited JST and file size distributions) in the parameter space values of the GROMACS input files.

The specific methodology of the new simulated workload is complicated, but is as follows:

1. For all of the pipeline stages, *except* `6.mdrun`'s JST, the simulation's average JST and file

size values come from the *S-peptide* input file (Table 5.2).

The actual JST of a job for all stages (including `6.mdrun`, below) within an instance is from a normal distribution in the range [(0.9 × average JST), (1.1 × average JST)].

Since it is part of the GROMACS software distribution, *S-peptide* is readily available. For completeness, the `6.mdrun` values for *S-peptide* are given in Table 5.2, but the JST=328 value is not used in the simulation.

2. For the JSTs for stage `6.mdrun`, the simulated workload uses a combination of values from *gmxbench* (Table 5.3). Specifically, for 100 workflow instances, 25 instances are based on each of the four benchmarks in *gmxbench*. Therefore, there are 25 instances with {average JST=5433 s, S-peptide file size=426 KB}, 25 instances with {average JST=708 s, S-peptide file size=426 KB}, 25 instances with {average JST=112 s, S-peptide file size=426 KB}, and 25 instances with {average JST=97 s, S-peptide file size=426 KB}. As with the other stages, the actual JST of the `6.mdrun` job within an instance is from a normal distribution around the average JST, as described above.

   Note that the JST and file sizes for pipeline stages 0 to 5, and 7, are not available for *d.dppc*, *d.lzm*, *d.poly-ch2*, and *d.villin* because the benchmarks are defined purely in terms of stage `6.mdrun`, with the required files for the other stages being unavailable.

3. In our experiment (Figure 5.5), all the data points are the averages of 10 runs by changing the random seed in the simulator (e.g., used to generate the normal distribution around the average JST), where the observed standard deviation of the simulated makespan (i.e., Y-axis) is very low.

The lack of some of the input files for *gmxbench* makes it necessary to base the simulation values primarily on *S-peptide*. But, in a desire for a variety of JSTs for `6.mdrun`, we use the WaFS-measured JSTs from *gmxbench*. Other methodologies for creating a WaFS-measured workload are possible; our workload still serves to validate the basic ideas behind WaFS, the prototype, and provides some new simulation results.

As the storage units in the budget varies from 2,000 KB to 100,000 KB, there are marked differences in simulated makespan between the different polices (Figure 5.5). Note that the most directly comparable results from earlier simulations are Figures 4.33(c) and 4.34(c). As expected, the absolute performance differences between the policies change given the different JST and file size parameters.

Pipeline Stage	Job Service Time (s)	File Size (KB)
0.pdb2gmx	8	49
1.editconf	5	9
2.genbox	16	146
3.grompp	10	469
4.mdrun	23	135
5.grompp	11	471
6.mdrun	328	426
7.g_energy	140	15

Table 5.2. WaFS-measured Job Service Times and File Sizes for the GROMACS *S-peptide* Workflow

Benchmark	Job Service Time (s)	File Size (MB)
d.dppc	5433	12
d.lzm	708	2
d.poly-ch2	112	2
d.villin	97	1

Table 5.3. WaFS-measured Job Service Times and File Sizes for the `6.mdrun` stage of GROMACS *gmxbench*

But, consistent with earlier results, DTO remains the best overall algorithm because it appropriately considers the file storage requirements of the later pipelines stages. DTO (still) has the lowest makespan, with the advantage of DTO over DAR, banker's algorithm, and Lang's algorithm growing as the storage budget increases to 100,000 KB. Although not shown, all of the policies converge in makespans as the storage budget gets even larger and deadlock is no longer a concern.

In particular, the file size requirement of stage `0.pdb2gmx` is low (i.e., 49 KB) relative to stages `3.grompp` (469 KB), `5.grompp` (471 KB), and `6.mdrun` (426 KB). Therefore, the tendency for banker's algorithm, Lang's algorithm and DAR to admit more workflow instances given a small initial resource requirement of 49 KB, often leads to blocked instances in the later stages, which results in high inactive resource utilization, which also results in higher makespans. But, DTO tends to give resources to already admitted instances, which allows the instances to complete faster instead of blocking.

Lang's algorithm, consistent with earlier simulations, still has the worst performance for the GROMACS workload. An interesting observation is that the detection algorithm shows a better performance than all the other algorithms, except for DTO. We observed the deadlock frequency to be high, but the released storage (after deadlock is detected and victim instances are killed) can allow the deadlocked instances, which are usually blocked on the data-intensive jobs, to finish their as quickly as possible.

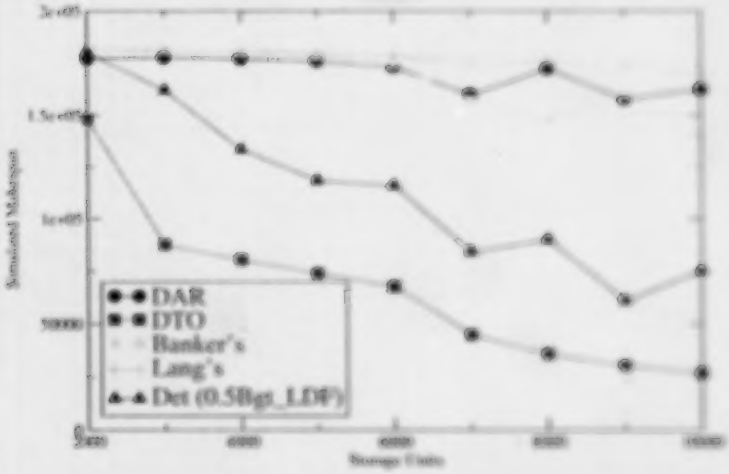


Figure 5.5. Simulation Results for the GROMACS Workload. Compared to Figure 4.33(c) and 4.34(c), DTO shows greater performance advantages compared to other policies.

5.5 Concluding Remarks

This chapter introduced our ptrace-based WolFS prototype with a special emphasis on the dataflow collection. To this end, we detailed how the `open()` and `close()` system calls are handled, file pathnames are manipulated and process family is tracked.

To validate the prototype, we used a real application workflow from GROMACS, called *fs-hemolysin 5-peptide*, as an example to illustrate how WolFS automatically collects the dataflow information. In addition, we also measured the performance overhead of WolFS for the GROMACS benchmark *gmxbench*. The overheads are measured to be between 0.39% and 11.74%, with lower overheads associated with longer job service times.

Finally, we used the gathered dataflow information of the *5 peptide* and *gmxbench* as the input/runtime data for a GROMACS workload to evaluate our algorithms. Our results are qualitatively consistent with earlier results, showing the advantages of the DTO algorithm for pipeline workflows.

Chapter 6

Related Work

We provide an overview of related work more or less according to our research scope. In Chapter 6.1 we describe some existing techniques to resolve filename conflicts in workflow computation. Some related file system studies are reviewed in Chapter 6.2. In high performance computing dataflow information can be used in various ways, and we survey some of its applications in Chapter 6.3. The most recent work in storage-aware workflow scheduling is discussed in Chapter 6.4. Finally, we review some related work on deadlock avoidance in Chapter 6.5.

6.1 Filename Conflict Resolution

To enable multiple workflow instances to execute concurrently, existing systems adopt different strategies to avoid filename conflicts. *Grid Execution Language* (GEL) [64] is a scripting language developed by the Bioinformatics Institute, Singapore, to facilitate job scheduling in grid computing systems. It allows multiple instances of the same workflow to execute concurrently by creating a working directory for each instance. All binaries (in each instance) run in the same working directory where they read, create and modify files based on the control-flow information. The output of the instance may finally be moved to some where from its working directory. However, using a separate directory in a control-flow-driven scheduler to isolate computations suffers from potentially higher storage overhead than necessary.

Unlike GEL, *DAGMan* [18] in *Condor* [94] (on which DAGMan is built) provides a number of complementary mechanisms to manage multiple instances of similar jobs and to help avoid filename conflicts. For output files Condor uses the $(Cluster) (i.e., job ID) macro when naming them so that they are unique to each job instance.

```
universe = vanilla
executable = /bin/hostname
output = results.$(Cluster)
error = errors.$(Cluster)
log = log.$(Cluster)
queue
```

Figure 6.1. An Example of Condor Submit Description File

For example, we could specify the job submit file shown in Figure 6.1 for multiple nodes in a DAG. This will create files like "`results.132`", "`results.133`", "`results.N`", etc., where `N` is the Condor job id of the actual job instance.

To help avoid conflicts in a job's runtime files, each Condor vanilla or standard universe job is executed in its own unique job "sandbox" directory on the remote execution host. As a result, while it runs, any runtime files used by the job (in the current working directory) are safe from other jobs' instances running beside the job. Although DAGMan can avoid the filename conflicts in concurrent executions of multiple instances of a *single* job, it is not clear how this mechanism can deal with a workload that is composed of multiple dependent jobs.

The most common solution to filename conflicts in batch schedulers is to execute each workflow instance in a sequential order (Figure 3.2(a), Serial Policy (BASE)). The serial policy is simple, but unlike the previous strategies it does not allow any *inter-workflow instance concurrency*. Despite the low degree of concurrency of the serial policy, its simplicity makes it a popular choice.

6.2 Related File System Studies

File systems have been studied in various computing environments for different workloads and with different goals. For example, *FileNet* [25] was designed to support a class of read-mostly workloads (e.g., document image processing) in a distributed system. *Zebra* [45] is a network file system that combines the two ideas of *a log structured file system* (LFS) and *striping with parity calculations* to increase file access throughput. *Elephant* [79, 80] is a versioning file system with a design goal of automatically retaining all important versions of a user's files. Recently, the *Google File System* (GFS) [36] was developed to address issues in fault tolerance, the management of large data sets and the optimization of append-intensive files for large distributed data-intensive applications. Unlike these file systems, our *Workflow-aware File System* (WaFS) is oriented to high-performance workflow-based workloads. It is designed to layer on top of the traditional file systems to discover the workflow-specific information automatically. And in terms of integrating the job scheduler and

file system, WaFS is similar to *BAD-FS* [12], a batch-aware distributed file system, but BAD-FS (at this time) is designed to deal with the issues of data consistency and replication but not scheduling. In contrast, the development of WaFS is motivated by a desire to improve job concurrency and to allow for efficient deadlock avoidance using dataflow information.

Linking File System (LiFS) [5] and *Transparent Result Caching* (TREC) [97] are also related to WaFS. LiFS extends the traditional set of file system metadata to include not only arbitrary, user-specified key-value pairs on files but also relationships between files in the form of links with attributes. Although LiFS can be used to record the dataflow information, it cannot discover such information automatically as the computation proceeds, unlike WaFS.

TREC is a general framework for transparently tracking process lineage (i.e., each process's parent, children, input files and output files) and file dependencies (i.e., for each file, the sequence of operations and the set of input files used to create the file). TREC can be used to deduce the dataflow information by observing program execution. However, it does not consider the filename conflict problem, and TREC does not use an integrated job scheduler and file system as WaFS does.

Versioning file systems are not a new idea. Traditionally, versioning file systems are designed to record the history of changes to files and to facilitate easy back-ups and rollbacks to previous versions of files. Some versioning file systems in the literature include Elephant [79,80], *Versionfs* [72], *Wayback* [19] and *Moraine* [103]. However, none of these systems are integrated with a batch scheduler with the purpose of improving job concurrency. In addition to per-file versioning, versioning techniques also include volume and file system *snapshots*. A snapshot is a read-only, logical image of a collection of data as it appeared at a single point in time. For example, the *ext3cow* [74] file system, which is built on Linux's popular *ext3* file system, takes advantage of snapshot capabilities to provide users with a time-shifting file system.

6.3 Dataflow Applications

The idea of exploiting dataflow information to facilitate computation is certainly not new. For example, in parallel computer architectures [7], dataflow concepts are used to overcome the difficulty of conventional control-flow-based architectures in maximizing instruction-level parallelism, which is roughly analogous to *intra-workflow instance concurrency* in our context. Of course, *Versioned Namespace* (VNS) and *Overwrite-Safe Concurrency* (OSC) use dataflow information *primarily* to improve inter-workflow instance concurrency (which could be compared to *Simultaneous Multithreading* (SMT) processor designs [26, 96]). Nonetheless, in the software systems context OSC

and VNS are unique in their ability to improve job scheduling through the integration of the file namespace manager (for gathering and controlling dataflow information) and the scheduler (for exploiting the dataflow information).

In compiler optimization, where dataflow and control-flow analyses are common, the compiler attempts to improve performance by increasing instruction-level parallelism and by re-ordering or transforming code to reduce overheads. Of course, OSC and VNS improve the degree of concurrency. However, neither of our proposed strategies attempt to re-order or transform the jobs of a workflow since there is as yet no higher-level semantics (e.g., a programming language for a compiler) to constrain and guide such transformations.

In high-performance computing systems dataflow is often used to improve job-level concurrencies. For example, *LSF* Batch [108] can define a job that is dependent on file events in advance so that a job can run after some file event (e.g., file arrival) has occurred. The *Workflow Enactment Engine* [105] proposes a decentralized event-driven scheduling architecture by using *tuple space* as a communication mechanism between data-dependent tasks. It allows tasks to be scheduled based on data dependency rather than waiting for the completion of (control-flow-based) parent jobs. A similar idea is also applied in *PAGIS* [101], a metacomputer system that uses *process networks* [62] as a semantic model for the composition of complex tasks in a geographic information system. In the metacomputer system a server takes a process network from a client and distributes work to the workers. The data is moved between the worker processes via a central queue in the server. Our work has the same purpose (i.e., improving concurrency) as these systems, however, our focus is on inferring the dataflow information automatically via the combination of the batch schedulers and file systems and further exploiting such information to maximize both intra- and inter-workflow instance concurrency.

In addition to maximizing job concurrency, dataflow information can be exploited for other purposes. For example, BAD-FS [12] refers to the flow of data as *I/O scoping* and uses it to compute an execution plan to minimize the network traffic. In *Kepler* [67], a scientific workflow system built upon the dataflow-oriented *Ptolemy II* system [75], dataflow information is used to specify the execution semantics of a workflow in a diagram via a *Process Network Director*. *MSF* [52] introduces a workflow service infrastructure for computational grid environments. Static dataflow information is described in its *Job Control Markup Language* (JCML) to facilitate data movement.

Dataflow information can be viewed as a special kind of application-specific information, which is exploited by our *WaFS Scheduler* to improve its scheduling. In this sense the WaFS Scheduler has the same philosophy as *AppLeS* [13, 14] and *MARS* [34].

AppLeS [13, 14] is an application-level scheduling project with a primary focus on developing scheduling agents for individual applications on production computational grids. The approach is to use application-specific information to model the application performance under a given set of resources (i.e., creating a performance model) and then based on the performance model to schedule the application. The programming model of AppLeS is on a per-application basis. To relieve programmers of the burden, AppLeS *Templates* were developed as software frameworks to embody common characteristics from various applications with similar structure and the same computational model.

MARS [34] is a framework for minimizing the execution time of distributed applications on a metacomputer. The application is usually structured as a *Single-Program-Multiple-Data* (SPMD) program that consists of multiple phases. For each phase, its execution profile remains the same across several runs. However, the phases are identified by users rather than detected automatically. MARS uses the phase profiling data of previous runs to derive an improved task-to-process mapping.

The differences between our WaFS Scheduler and these systems are that the WaFS Scheduler utilizes the application-specific information (i.e., dataflow information) via the underlying file system (i.e., WaFS), and such information is automatically discovered by WaFS, whereas, the development of AppLeS templates and identification of MARS phases are all the users' responsibilities.

6.4 Storage-Aware Workflow Scheduling

The interest in scheduling workflow-based computations in storage-constrained HPC systems is increasing with the awareness of the growth of datasets [9,41]. In this section we review two storage-aware workflow scheduling systems that have recently been published in the literature [12, 76]. Comparison with our WaFS Scheduler are given in Table 6.1.

Bent *et. al* proposed a *capacity-aware scheduling* in BAD-FS [12], in which a centralized batch scheduler manages the storage space by carefully allocating storage volumes for the jobs from multiple workflow instances so that storage overflowing or cache thrashing can be avoided. To achieve these ends, Bent *et. al* identified five possible data allocation strategies which influence the execution path and the performance of the workloads [11]. Although these allocation strategies can *prevent* deadlocks (not stated explicitly by the authors), they are unable to make the best use of storage resources for performance optimization. For example, neither the *AllPrivate* strategy nor the *AllBatch* strategy optimizes the allocation of the available storage. On the other hand, all their allocation strategies are designed for batch-pipeline workflows and might not be effective for other

System	Workflow Shape	Workflow Instance	Storage Site	Garbage Collection	Deadlock
Bent's [12]	Pipeline	Multiple	Single	Not Clear	Prevention
Ramakrishnan's [76]	Arbitrary	Single	Multiple	Cleanup Job	Ostrich
WaFS Scheduler	Arbitrary	Multiple	Single	Scheduler Control	Avoidance

Table 6.1. Comparison between Bent's System, Ramakrishnan's System and the WaFS Scheduler

workflow shapes such as those have examined in this thesis. Unlike the strategies designed to carefully allocate the storage, our alternative is to use a deadlock avoidance approach to deal with the storage constraints.

Most recently, Ramakrishnan *et. al* considered the scheduling of data-intensive workflows with more general shapes onto a set of storage-constrained distributed computational sites [76]. They address exactly the same problem as ours, i.e., improving the workflow data storage utilization. Their basic approach is to add a *cleanup job* for each data file when that file is no longer needed by other jobs in the workflow or when it has already been staged out to some permanent storage. The garbage files are deleted (also called *garbage collection*) in time, and the amount of storage used for the workflow can be reduced significantly. Although the cleanup jobs are not compute-intensive, the large number of cleanup jobs may cause performance degradation. To mitigate this problem, they also implemented a heuristic that uses a single cleanup job for removing multiple files. Unlike their approach, the garbage collection in our system is directly controlled by the batch scheduler based on the dataflow information gathered in WaFS, rather than requiring cleanup jobs.

A major difference between our system and theirs is how deadlock is dealt with. Their algorithm is storage-aware in the sense that when deciding to schedule a job, the disk space available from each site is first considered and only the eligible sites (i.e., the sites with sufficient disk space) are prioritized according to some performance metrics for the job scheduling. Job are mapped to the sites with the highest priority first to minimize the overall execution time of the workflow. However, deadlock is not considered in their algorithm. When a deadlock occurs, the workflow is simply aborted. This might not be a serious problem in the situation where a single workflow instance is scheduled onto multiple storage sites. We consider scheduling multiple workflow instances onto a single storage site, which is similar to the situation in Bent's work, and deadlock is usually a pragmatic concern.

Algorithm	Time Complexity	Major Techniques	Comments
Habermann [43]	$O(mn^2)$		Not all processes need to be sequenced in the safety check
Holt [49]	$O(mn^2)$	a time-based technique	address the artificial deadlock
Minoura [70]		(1) resource-request graph, but the resource requests are made in a nested form. (2) localized maximum claims are computed by dynamic may-wait for graph.	localized maximum claims
Lang [60]		resource-request graph and decomposition into regions	localized maximum claims
Kameda [56]	$O(mn^{\frac{3}{2}})$	network technique	
Belik [10]	$O(mn)$	safe sequence reduction	amortized worst case
Habermann [44]	$O(r)$		single resource type
Finkel&Madduri [31]	$O(logn)$	allocation history in a binary tree	single resource type
Lee [63]	$O(n)$	parallel algorithm and hardware implementation	for MPSoC

Table 6.2. Comparison of Some Banker's-based Deadlock Avoidance Algorithms (m: the number of resource types, n: the number of processes, r: the number of resource units)

6.5 Deadlock Avoidance

Deadlock avoidance attempts to keep the system in a set of safe states, where the circular chain of resource contention that produces the deadlock cannot occur. To achieve this, it is usually necessary to have some advance information about the resource use of processes. For example, the most widely recognized banker's algorithm requires a priori knowledge of the maximum amount of resources needed by each process. Unfortunately, such knowledge is not always available in reality, rendering this algorithm mostly inapplicable in practice. However, as previously discussed, in workflow-based computations the storage requirements of each job are in general available, making the deadlock avoidance algorithm a promising approach.

The banker's algorithm, since it was originally proposed by Dijkstra to handle a single resource type [23], has initiated much follow-up research, most conducted in the early days of operating systems [43,46,49,50,70]. However, the banker's algorithm still forms the basis for many deadlock avoidance algorithms in a variety of application contexts [8,10,60,61]. In the studies on the banker's algorithm the research is mainly concentrated on the resource utilization and time complexity of the algorithms. We summarize some related work in this area in Table 6.2, where the first three studies focus on improving the resource utilization, whereas the last five concentrate on minimizing the time complexity. Our work largely falls into the former class, since for workflow-based workloads the execution time of the constituent jobs is generally much longer than the deadlock resolution time.

The complexity of the deadlock avoidance algorithm is thus not our primary concern.

In order to improve the resource utilization, Habermann redesigned the banker's algorithm and extended it to multiple serially reusable resource types [43]. Although the contribution was important in making the banker's algorithm more general, it could not address a practical problem of *permanent blocking*, identified by R. Holt [49] as the scheduler-incurred *artificial deadlock* (i.e., the processes with safe requests were never scheduled). This problem was discussed by Holt and addressed by a time-based technique.

In addition to the extension of the functionality, other research efforts focused on refining the algorithm based on some interesting process models, each differing in the amount of information that is assumed to be available [77, 85]. The purpose of these works is primarily to minimize the conservativeness in the safety check so that the resource utilization of the algorithms can be improved. An early effort was made by Minoura [70] with the concept that the control-flow of the involved processes can be modeled as a *resource-request graph*, which is a rooted tree of nodes, each node representing either a resource request or a resource release. Based on the resource-request graph, the current execution points of a process can be tracked, and then the *localized approximate maximum claim* of the process can be computed by using a *dynamic may-wait-for graph*. Finally, a modified version of Habermann's algorithm leverages this localized maximum claim to improve upon the original algorithm. However, this algorithm has a major limitation that it requires the resource requests to be made in such a way that for each resource type within a process, the units granted last are released first (i.e., a "nested" form).

Later, based on the same resource-request graph, Sheau-Dong Lang [60] described a natural extension of the banker's algorithm to overcome the limitation of the modified Habermann's algorithm in [70]. Lang's algorithm decomposes the control-flow graph of a process into a nested family of *regions* and improves upon the banker's algorithm by having the knowledge of the localized approximate maximum claim associated with each region. Lang's algorithm does not require a "nested" form of resource-request graph. However, it still inherits the rooted-tree-like resource-request graph, which, although it represents a wide range of applications' control-flows, cannot effectively model the storage access patterns in our workflow-based computations.

The complexity of the banker's algorithm and its variants is also a major concern in practice. Given n processes and m resource types, Habermann's algorithm requires $O(mn^2)$ time for the safety check. Kameda [56] presented an algorithm which with an aid of a network technique, reduces the complexity to $O(mn^{\frac{3}{2}})$. Belik [10] modified the banker's algorithm by slightly sacrificing the resource utilization but achieving an $O(mn)$ amortized worst case running time under certain

likely conditions. When considering a single resource type, Habermann [44] proposed an efficient algorithm to handle a resource request or release within the space and time complexity of $O(n + r)$ and $O(r)$, respectively, where r is the number of resource units. Later, the time complexity was improved to $O(logn)$ by Finkel and Madduri [31], whose algorithm maintains the resource allocation history in a binary tree. More recently, Lee [63] proposed a $O(n)$ parallel banker's algorithm and implemented it in hardware to provide a mechanism for very fast, automatic deadlock avoidance for a MultiProcessor System-on-a-Chip (MPSoC).

In addition to operating systems, the banker's algorithm is also applicable in other areas where deadlock is recognized as a serious problem. For example, Lomet [66] tailored the algorithm to the needs of database systems. The proposed algorithm is not only simpler but also performs better in a database environment. In addition, the algorithm provides additional functions that are absent in the earlier algorithms. The banker's algorithm was also refined to deal with the deadlock problems in *Flexible Manufacturing Systems* (FMS) [8, 61, 77], where new process models and more static knowledge are always available to maximize concurrency while avoiding the deadlocks.

When compared with this related work, our algorithms, *Dataflow-based Aggregate Requests* (DAR) and *Dataflow-based Topological Ordering* (DTO), of course, bear certain similarities to some existing algorithms [60, 77]. However, our algorithms are unique in that they make a distinction between active and inactive resources for makespan reduction, which is not a part of previous approaches.

Chapter 7

Concluding Remarks

7.1 Limitations

Our proposed system provides the benefits of transparently detecting the dataflow information and exploiting it to efficiently overcome the artificial constraints on concurrency in most current batch schedulers. However, our system does has some limitations:

1. Dataflow graphs are assumed to be static from instance to instance. Although this assumption seems restrictive, we believe it is quite reasonable in scientific computation.

2. It is assumed that no filename conflicts occur inside workflow instances. In other words, filename conflicts are only possible between workflow instances. In general, it is difficult to resolve filename conflicts inside workflow instances due to the associated race conditions. Such conflicts effectively represent specification errors and hence are not considered.

3. It is assumed that the job service time and file sizes are always over-estimated; otherwise, deadlock can occur.

4. Our simulation assumes a centralized batch scheduler and a single storage-constrained site. However, we believe that our system can be generalized to other configurations.

5. Our WaFS prototype is not a full, production-quality system. Instead, it is just a proof-of-concept implementation used to validate our basic design and show one possible implementation strategy. Any implementation or architecture that reliably gathers dataflow information can be used with our scheduling policies.

Although some of the described limitations prevent our system from dealing with a number of use cases, such cases are believed to be rare in practice; hence, our system is useful in spite of these limitations.

7.2 Conclusions

The research in this thesis was motivated by some specific problems (limitations) of control-flow-driven batch schedulers used in current HPC systems with respect to filename conflicts and deadlock in storage allocation. Although these problems are not fundamental to batch schedulers, filename conflicts and deadlock impose constraints on job scheduling that limit the degree of concurrency and lower the efficient utilization of storage resources.

Our major contribution in this thesis is in demonstrating the value of dataflow information in addressing these problems and advocating for a systematic solution that is more transparent, easier to use and has performance benefits. We proposed a system (i.e., the *WaFS Scheduler*) based on an integrated file system and batch scheduler. The essence of the system is to extend a traditional file system into a *Workflow-aware File System* (WaFS) for capturing and managing the dataflow information as the computation proceeds. The system then makes this gathered dataflow information available to the batch scheduler in order to maximize job concurrency given the filename conflicts and reducing the impact of deadlock when storage resources are limited, which are generally not possible in the traditional control-flow-based batch schedulers.

To leverage the dataflow information, we have developed and evaluated three scheduling policies, *Versioned Namespace* (VNS), *Overwrite-Safe Concurrency* (OSC) and their *hybrid* (HB), through simulation studies.

By combining dataflow information with a versioned namespace, we showed that VNS can reduce the makespans by over an order-of-magnitude, depending on the arrival rate of the workload, while the storage overhead is low.

In contrast, OSC takes advantage of dataflow information to safely overwrite files instead of always versioning files as per VNS. Thus, OSC is able to minimize the storage overhead but at the expense of losing DOC when more storage resources are available. Both policies exhibit advantages over the traditional sub-directory-based resolution of filename conflicts in terms of lower makespan and storage overhead.

To combine the advantages of versioning and overwriting, HB was also studied for its ability to maximize both DOC and efficient storage resource utilization. The key point of HB is to leverage the

inferred dataflow information to effectively resolve the deadlock problem when multiple concurrent workflow instances compete for the limited available storage resources.

To this end, we proposed two deadlock avoidance algorithms, *Dataflow-based Aggregate Requests* (DAR) and *Dataflow-based Topological Ordering* (DTO), based on the classic banker's algorithm. However, unlike previous studies, which generally do not distinguish between *active* and *inactive* resource utilization, our algorithms leverage the dataflow information to make the important distinction between active and inactive resources for makespan reduction. A key part of that distinction is to compute maximum resource claims dynamically (as with DAR and DTO), instead of statically (as with the banker's algorithm).

First, with DAR the maximum claim associated with each instance is computed at runtime by using the dataflow information to sum the storage requirements of all the remaining jobs. Second, in DTO the dataflow knowledge is exploited to topologically order the remaining jobs in the current instance when checking for safety. Both algorithms try to maximize the active storage utilization by improving either the inter-instance concurrency or the intra-instance concurrency. Our simulation studies show that the proposed algorithms, in most cases, are better than the static, control-flow-based banker's algorithm and Lang's algorithm in terms of both makespan and active storage utilization.

7.3 Future Work

A major future research direction is to take the WaFS prototype (Chapter 5) and expand it into a production-level system. The benefits of our new algorithms, as validated by the simulation-based study of this dissertation, provide significant motivation to proceed on that system-building line of research. Furthermore, real applications (in addition to GROMACS) can be, and should be, studied.

A natural and worthwhile extension of the existing simulation-based studies is to consider more workflow shapes and different random number distributions, beyond uniform and Zipf distributions (Appendix B).

Lastly, in terms of scheduling algorithms, the emphasis of this dissertation has been on minimizing makespan. However, mean response time (MRT) is another key metric in scheduling and a proper study of using dataflow information to minimize MRT would also be valuable.

Bibliography

[1] T.L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, 1974.

[2] A.D. Alexandrov, M. Ibel, K. E. Schauser, and C.J. Scheiman. Ufo: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, 16(3):207–233, 1998.

[3] M.P. Allen. Introduction to molecular dynamics simulation. In N. Attig, K. Binder, H. Grubmuller, and K. Kremer, editors, *Computational Soft Matter: From Synthetic Polymers to Proteins, Lecture Notes*. Gustav-Stresemann-Institut, Bonn, Germany, 2004.

[4] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[5] A. Ames, N. Bobb, S.A. Brand, A. Hiat, C. Maltzahn, E. L. Miller, A. Neeman, and D. Tuteja. Richer file system metadata using links and attributes. In *Mass Storage Systems Technologies (MSST2005)*, Monterey, CA, 2005.

[6] NASA Ames and the Courant Institute at NYU. Cart3D, `http://people.nas.nasa.gov/~aftosmis/cart3d/cart3Dhome.html`.

[7] K. Arvind and R.S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3), March 1990.

[8] Z.A. Banaszak and B.H. Krogh. Deadlock avoidance in flexible manufacturing systems with concurrently competing process flow. *IEEE Transactions on Robotics and Automation*, 41(6):724–734, 1990.

[9] B.C. Barish and R. Weiss. Ligo and the detection of gravitational waves. *Physics Today*, 52, 1999.

[10] F. Belik. Deadlock avoidance with a modified banker's algorithm. *BIT*, 27(3):290–305, 1987.

[11] J. Bent. Data-driven batch scheduling, 2005. Ph.D thesis, University of Wisconsin-Madison.

[12] J. Bent, D. Thain, A.C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of Networked Systems Design and Implementation (NSDI)*, pages 365–378, San Francisco, California, USA, 2004.

[13] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, S. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the grid using apples. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003.

[14] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application level scheduling on distributed heterogeneous networks. In *Proceedings of the ACM/IEEE conference on Supercomputing*, Pittsburgh, PA, November 1996.

[11] P. Blaha, K. Schwarz, G. Madsen, D. Kvasnicka, and J. Luitz. WIEN2k: An augmented plane wave plus local orbitals program for calculating crystal properties. Technical report, Institute of Physical and Theoretical Chemistry, Vienna University of Technology, 2001.

[16] J. Blythe, Y. Gil, and E. Deelman. Combining workflows in shared grid environments. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling*, Whistler, British Columbia, Canada, 2008.

[17] E.G. Coffman. *Computer and Job-Shop Scheduling Theory*. Wiley, New York, 1976.

[18] Condor Team, 2008. http://www.cs.wisc.edu/condor/dagman.

[19] B. Cornell, P. Dinda, and F. Bustamante. A user-level accounting file system for Linux. In *Proceedings of CLUSTER*, Boston, MA, USA, 2004.

[20] P.E. Crandall, R.A. Aydt, A.A. Chien, and D.A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, pages 59–89, San Diego, California, USA, 1995.

[21] Cray. http://www.cray.com/products/software/rapt.

[22] L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986.

[23] E.W. Dijkstra. *Cooperating Sequential Processes*. New York Academic Press, 1968.

[24] R.P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51:161–166, 1950.

[25] D.A. Edwards and M.S. Mckendry. Exploiting read-mostly workloads in the Gluon file system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 58–70, Litchfield Park, Arizona, USA, 1989.

[26] S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, R.L. Stamm, and D.M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997.

[27] M. El-Rewini, T.G. Lewis, and H.H. Ali. *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., 1994.

[28] B. Abbott et. al. Search for gravitational waves from binary inspirals in S3 and S4 LIGO data. http://arXiv.org/abs/[illegible].

[29] D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing (IPPS'97)*, pages 1–34, 1997. Also published as Springer Verlag LNCS 1291.

[30] D.G. Feitelson and M.A. Jette. Improved utilization and responsiveness with gang scheduling. In *3rd Workshop on Job Scheduling Strategies for Parallel Processing (IPPS'97)*, pages 238–261, 1997.

[31] R. Finkel and G.M. Muldner. An efficient deadlock avoidance algorithm. *Information Processing Letters*, 24(1):25–30, January 1987.

[32] D.K. Friesen. Tighter bound for LPT scheduling on uniform processors. *SIAM Journal on Computing*, 16(3):554–560, June 1987.

[33] P. Obarzynski. [illegible]. http://www.cs.caltech.edu/~paweo/[illegible]-enough.html.

[34] J. Gehring and A. Reinefeld. MARS-A framework for minimizing the job execution time in a metacomputing environment. *Proceedings of Future General Computer Systems*, 12(1):87–99, 1996.

[35] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16:276–291, 1992.

[36] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–43, New York, USA, 2003.

[37] R. Gibbons. A historical application profiler for use by parallel schedulers. In *3rd Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 58–77, Geneva, Switzerland, 1997.

[38] T. Glatard, J. Montagnat, and X. Pennec. Grid-enabled workflows for data intensive medical applications. In *18th IEEE Symposium on Computer-Based Medical Systems*, pages 537–542, Trinity College Dublin, Ireland, 2005.

[39] Genias Software GmbH. CODINE: Computing in distributed networked environments, 1995.

[40] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[41] J. Gray, D.T. Liu, M. Nieto-Santisteban, A. S. Szalay, D. DeWitt, and G. Heber. Scientific data management in the coming decade. Technical Report MSR-TR-2005-10, Microsoft Corporation, 2005.

[42] GROMACS. `http://www.gromacs.org`.

[43] A.N. Habermann. Prevention of system deadlocks. *Communication of ACM*, 12(7):373–385, July 1969.

[44] A.N. Habermann. *Introduction to Operating System Design*. Chicago:Science Research Association, Inc., 1976.

[45] J.H. Hartman and J.K. Ousterhout. The Zebra striped network file system. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 309–329. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[46] J.W. Havender. Avoiding deadlock in multitasking systems. *IBM System Journal*, 2:74–84, 1968.

[47] R. Henderson and D. Tweten. Portable batch system: External reference specification, 1996. NASA Ames Research Center.

[48] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach, Third Edition*. Morgan Kaufmann, 2005.

[49] R.C. Holt. Comments on prevention of system deadlocks. *Communications of ACM*, 14(1):36–38, January 1971.

[50] R.C. Holt. Some deadlock properties of computer systems. *ACM SIGOPS Operating Systems Review*, 6:64–71, June 1972.

[51] P. Hulith. The AMANDA experiment. In *Proceedings of the XVII International Conference on Neutrino Physics and Astrophysics*, Helsinki, Finland, June 1996.

[52] S. Hwang and J. Choi. MSF: A workflow service infrastructure for computational grid environments. In *Workshop on Grid Computing and its Application to Data Analysis (GADA)*, volume 3292, pages 222–231, 2004. In conjunction with OnTheMove Federated Conferences (OTM), and published in Lecture Notes in Computer Science.

[53] IBM. Licensed program specifications: IBM LoadLeveler version 1 release 3.0, 1996. IBM Document GH23-0040-03.

[54] D. Jackson, Q. Snell, and M. Clement. Core algorithms of the Maui scheduler. In *7th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 87–102, Cambridge, MA, USA, June 2001.

[55] M.J. Gonzalez Jr. Deterministic processor scheduling. *ACM Computing Survey*, 9(3):173–204, September 1977.

[56] T. Kameda. Testing deadlock-freedom of computer systems. *Journal of the ACM*, 27(2):270–280, April 1980.

[57] Y.-K. Kwok. Dynamic critical-path scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[58] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Survey*, 31(4):406–471, September 1999.

[59] R. Lake, J. Schaeffer, and P. Lu. Solving large retrograde analysis problems using a network of workstations. *Advances in Computer Chess*, VII:135–162, 1994.

[60] S.-D. Lang. An extended banker's algorithm for deadlock avoidance. *IEEE Transactions on Software Engineering*, 25(3):428–432, May/June 1999.

[61] M. Lawley, S. Reveliotis, and P. Ferreira. The application and evaluation of banker's algorithm for deadlock-free buffer space allocation in flexible manufacturing systems. *The International Journal of Flexible Manufacturing Systems*, 10:73–100, 1998.

[62] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[63] J.J. Lee and V.J. Mooney. A novel O(n) parallel banker's algorithm for system-on-a-chip. In *Proceedings of the 12th Asia and South Pacific Design Automation Conference*, Shanghai, China, January 2005.

[64] C.C. Lian, F. Tang, P. Issac, and A. Krishnan. GEL: Grid execution language. *Journal of Parallel and Distributed Computing*, 65:857–869, 2005.

[65] D. Lifka. The ANL/IBM SP scheduling system. In *1st Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 295–303, Santa Barbara, CA, USA, 1995.

[66] D.B. Lomet. A practical deadlock avoidance algorithm for data base systems. In *Proceedings of the 1977 ACM SIGMOD international conference on Management of data*, pages 122–127, Toronto, Ontario, Canada, 1977.

[67] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows*, 2005.

[68] C. Macdonell. Trigger scripts for extensible file systems, 2002. Master's thesis, University of Alberta, Canada.

[69] A. Mandal, K. Kennedy, C. Koelbel, B. Liu, and L. Johnsson. Scheduling strategies for mapping application workflows onto the grid. In *Proceedings of the 14th International Symposium on High Performance Distributed Computing (HPDC)*, pages 125–134, Research Triangle Park, NC, USA, July 2005.

[70] T. Minoura. Deadlock avoidance revisited. *Journal of the ACM*, 29(4):1023–1048, October 1982.

[71] S.S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann, 1997.

[72] K.-K. Muniswamy-Reddy, C.P. Wright, A. Himmer, and E. Zodok. A versatile and user-oriented versioning file system. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST)*, pages 115–128, San Francisco, California, USA, 2004.

[73] S. Nicolau, X. Pennec, L. Soler, and N. Ayache. Evaluation of a new 3D/2D registration criterion for liver radio-frequencies guided by augmented reality. In *International Symposium on Surgery Simulation and Soft Tissue Model*, pages 270–283, in-Les-Pins, France, June 2003.

[74] Z. Peterson and R. Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 5(2):190–212, 2005.

[75] PTOLEMY II project and system, 2004. `http://ptolemy.eecs.berkeley.edu/ptolemyII/`.

[76] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Mayers, and M. Samidi. Scheduling data-intensive workflows onto storage-constrained distributed resources. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid*, pages 401–409, 2007.

[77] S.A. Reveliotis and M. A. Lawley. Efficient implementations of banker's algorithm for deadlock avoidance in flexible manufacturing systems. In *Proceedings of the 6th International Conference on Emerging Technologies and Factory Automation*, Los Angeles, CA, USA, March 1997.

[78] A.L. Rosenberg. On scheduling mesh-structured computations for internet-based computing. *IEEE Transactions on Computers*, 53(9):1176–1186, September 2004.

[79] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 110–123, Kiawah Island Resort, near Charleston, SC, USA, 1999.

[80] D.J. Santry, M.J. Feeley, N.C. Hutchinson, and A.C. Veitch. Elephant: The file system that never forgets. In *Workshop on Hot Topics in Operating Systems*, pages 2–7, Rio Rico, Arizona, USA, 1999.

[81] J. Schaeffer and R. Lake. Solving the game of checkers. In Richard J. Nowakowski, editor, *Games of No Chance*, volume 20. Cambridge University Press, 1996.

[82] M. Schmidt, K. Baldridge, J. Boatz, S. Elbert, M. Gordon, J. Jensen, S. Koseki, N. Matsunaga, and J. Montgomery. The general atomic and molecular electronic structure system. *Journal of Computational Chemistry*, 14:1347–1363, 1993. `http://www.msg.ameslab.gov/GAMESS/GAMESS.html`.

[83] R. Sethi. Complete register allocation problem. *SIAM Journal on Computing*, 3(3):226–248, 1975.

[84] K.C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19(2-3):107–140, 1994.

[85] A. Shoshani and E.G. Coffman. Sequencing tasks in multi-process, multiple resource systems to avoid deadlocks. In *Proceedings of the 11th Annual Symposium on Switching and Automata Theory*, pages 225–233, October 1970.

[86] G.C. Sih and E.A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):75–87, Feb. 1993.

[87] D. Sima, T. Fountain, and P. Kacsuk. *Advanced Computer Architectures, A Design Space Approach*. Addison Wesley, 1997.

[88] A. Sulistio and R. Buyya. A time optimization algorithm for scheduling bag-of-task applications in auction-based proportional share systems. In *Proceedings of the 17th International Symposium on Computer Architecture and High Performance Computing*, pages 235–242, Rio de Janeiro, Brazil, October 2005.

[89] A.K. Sum and J.J. de Pablo. Nautilus: Molecular simulation code. Technical report, University of Wisconsin-Madison, Department of Chemical Engineering, 2002.

[90] Sun Microsystems. `http://www.sun.com/software/gridware/`.

[91] D. Szafron, P. Lu, R. Greiner, D.S. Wishart, B. Poulin, R. Eisner, Z. Lu, J. Anvik, C. Macdonell, A. Fyshe, and D. Meeuwis. Proteome analyst: Custom predictions with explanations in a web-based tool for high-throughput proteome annotations. *Nucleic Acids Research*, 32:W365–W371, 7 2004. `http://www.cs.ualberta.ca/~bioinfo/PA/`.

[92] A.S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2001.

[93] I. Taylor, M. Shields, I. Wang, and O. Rana. Triana applications within grid computing and peer to peer environments. *Journal of Grid Computing*, 1:199–217, 2004.

[94] D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, Inc., 2003.

[95] S.-A.-A. Touati. DDG, `http://www.prism.uvsq.fr/~touati/sw/DDG/`.

[96] T. Ungerer, B. Robic, and J. Silc. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35(1):29–63, 2003.

[97] A. Vahdat and T. Anderson. Transparent result caching. In *Proceedings of the USENIX Technical Conference, New Orleans, Louisiana*, 1998.

[98] J. Wang, H. Kuehl, and M.D. Sacchi. Least-squares wave-equation AVP imaging of 3D common azimuth data. In *Proceedings of the 73rd Annual International Meeting, Society of Exploration Geophysicists*, 2003.

[99] Y. Wang and P. Lu. On the benefits of a workflow-aware versioning filesystem in metacomputing systems. In *The 8th International Conference on High Performance Computing in Asia Pacific Region, Beijing, China*, pages 227–234, 2005.

[100] Y. Wang and P. Lu. Using dataflow information to improve inter-workflow instance concurrency. In *6th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 1078–1082, Dalian, China, 2005.

[101] D. Webb, A. Wendelborn, and K. Maciunas. Process networks as a high-level notation for metacomputing. In *International Workshop on Java for Parallel and Distributed Computing*, San Juan, Puerto Rico, 1999. In association with IPPS/SPDP 1999, and published in Lecture Notes in Computer Science, volume 1586.

[102] T. Werner. Target gene identification from expression array data by promoter analysis. *Biomolecular Engineering*, 17:87–94, 2001.

[103] T. Yamamoto, M. Matsushita, and K. Inoue. Accumulative versioning file system Moraine and its application to metrics environment MAME. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 80–87, 2000.

[104] T. Yang. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.

[105] J. Yu and R. Buyya. A novel architecture for realizing grid workflow using tuple spaces. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 119–128, Pittsburgh, USA, 2004.

[106] Z. Yu and W. Shi. An adaptive rescheduling strategy for grid workflow applications. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, pages 214–220, Long Beach, CA, USA, September 2007.

[107] Y. Zhang, C. Koelbel, and K. Kennedy. Relative performance of scheduling algorithms in grid environment. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid*, Rio de Janeiro, Brazil, 2007.

[108] S. Zhou. LSF: Load sharing in large-scale heterogeneous distributed systems. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, Florida, USA, December 1992.

Appendix A

Comparisons with Lang's algorithm in Workflow-Based Computation

In spirit, our algorithms bear some similarities to Lang's algorithm [60] where the *localized approximate maximum claims*, as opposed to the global maximum claims of each process, are used for the safety check. Thus, in this appendix, we will make a direct comparison between our algorithms and Lang's algorithm. To this end, we first review Lang's algorithm and identify its problems when applying it to the workflow-based computation and then describe how our algorithms address these problems.

A.1 An Overview of Lang's Algorithm

Lang's algorithm [60] is a natural extension of the banker's algorithm, with the aim of improving the potential of resource utilization while incurring low overhead. More specifically, Lang's algorithm models the control-flow of a process as a *resource-request graph*, represented as a rooted tree of nodes with each node corresponding to a resource request or a resource release. Such a graph is further decomposed into a nested family of *prime regions*. A prime region is defined as a directed path (i.e., a sequence of connected nodes) which satisfies the following conditions[1]:

1. No resources are allocated before the control enters the first node of the path.
2. All allocated resources are released when the control leaves the last node.
3. No proper subpaths also satisfy the first two properties.

[1] This definition does not require the resource requests be made in a nested form.

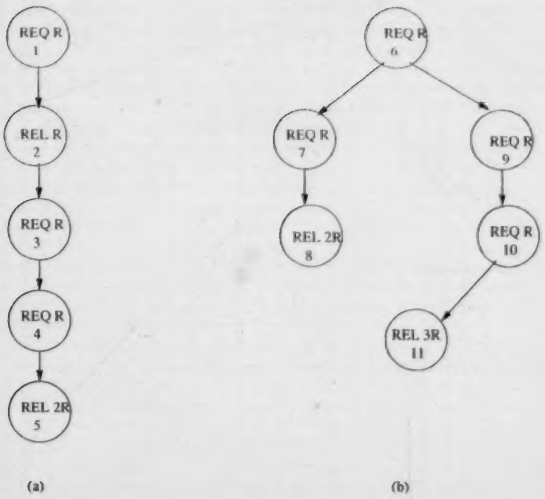


Figure A.1. Resource-request Graphs used in Lang's Algorithm: To compare with the storage utilization in workflow-based computation, only a single type of resources is depicted. In fact, Lang's algorithm can handle multiple types of resources.

A *subprime region* is a subpath of a prime region having the same end node. Some example resource-request graphs and their regions are shown in Figure A.1. For simplicity, only one resource type is illustrated. In Figure A.1(a), the path $\{1, 2, 3, 4, 5\}$ consists of two prime regions $\{1, 2\}$ and $\{3, 4, 5\}$, while in Figure A.1(b), $\{6, 7, 8\}$ and $\{6, 9, 10, 11\}$ are two prime regions. An example of a subprime region is $\{9, 10, 11\}$, which is a subpath of the prime region $\{6, 9, 10, 11\}$.

Based on the concept of region, Lang's algorithm avoids the deadlock in such a way that,

1. the information on the maximum resource claims for each region, i.e., *localized approximate maximum claims*, can be extracted prior to process execution.

2. when entering a new region of each process at runtime, the original banker's algorithm is applied by using the localized approximate maximum claims of the region instead of the global maximum claims of the whole control-flow graph.

3. all allocated resources are released before the control leaves a region.

The key point of Lang's algorithm is to determine for each node of a resource-request graph the corresponding region and the maximum resource claim associated with that region. This is achieved by the *Region Decomposition Algorithm*. In this algorithm a region for a node u, denoted by *region(u)*, is identified by collecting the nodes (including u) along the depth-first traversal path of the

resource-request graph until the net resources occupied by the collected nodes become zero. The maximum resource claim associated with the node u is pre-computed as well by the union of the claims associated with the prime or subprime regions in *region(u)*.

The details of Lang's algorithm can be found by referring to his paper [60]. Lang's algorithm suffers from the limitation of the rooted tree structure of the control-flow graph. Consequently, it cannot be directly used in deadlock avoidance for workflow-based computation, where the resource-request graph can be arbitrary shape.

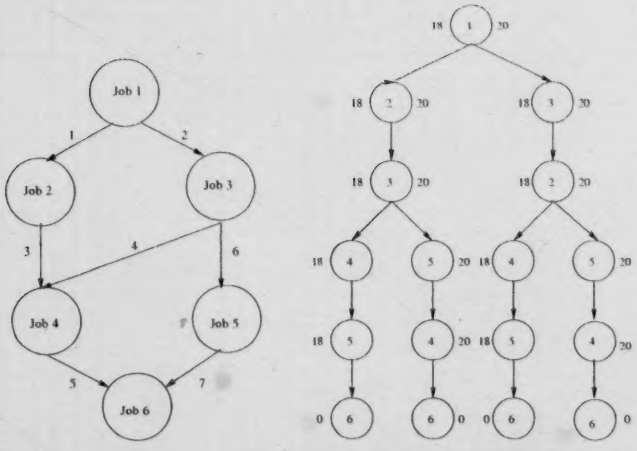
A.2 The Problems of Lang's Algorithm in Workflow-based Computation

When applying Lang's algorithm to a workflow-based computation, we first need to transform the dataflow DAG of the workflow into a resource-request graph by splitting each node in the workflow graph into a pair of nodes to represent the resource request and release, respectively. However, the resulting resource-request graph may have an arbitrary structure rather than the tree-like structure required by Lang's algorithm. Specifically, to process a resource-request graph with a more general structure, Lang's algorithm faces two major problems:

1. **When to compute:** In a tree-like graph, the scheduling order of a node is fixed (depending on the control-flow). Thus, the maximum claim associated with a node, according to Lang's algorithm, can be computed in advance. However, in a general graph the scheduling order of a node is generally unknown in advance (depending on the scheduling algorithm). Lang's algorithm cannot effectively compute the maximum claim for each node.

2. **How to compute:** In Lang's algorithm the localized maximum claim associated with a node is recursively computed by enumerating and traveling all its branches (rooted in the examined node) to select the largest maximum claim associated with its child node. The nodes along the path are executed sequentially. However, this computation procedure is impractical to compute the maximum claim associated with a node in a DAG since the scheduling space is generally intractable. In other words, computing the maximum claim for a node by recursively computing the maximum claims of its child nodes (i.e., those nodes which can be scheduled immediately after the examined node) in a general DAG is not effective.

To illustrate these problems, we use an example workflow DAG, shown in Figure A.2, where its scheduling graph is also shown. The scheduling graph is defined as a directed tree where the nodes along a path signify a scheduling order. To see the first problem, we can examine `Node 3`



(a) An Example of Workflow Graph

(b) The Corresponding Scheduling Graph: The associated maximum claims are marked beside the nodes

Figure A.2. An Example of Workflow DAG and its Scheduling Graph

in the scheduling graph. Depending on the unknown scheduling orders of the remaining nodes, the maximum claim associated with `Node 3` might be different (i.e., 18 or 20). Thus, Lang's algorithm cannot compute the maximum claim in advance.

The second problem is in computing the maximum claim for `Node 3` after `Node 1` has been completed and `Node 2` has been scheduled. In this small example, `Node 3` has two branches, with each branch having different maximum claims of 18 and 20, respectively. However, for a large graph, the *huge* number of branches might render the computation procedure for the maximum claims defined in Lang's algorithm to be intractable.

Although the scheduling graph is a tree, the potential size of the tree (i.e., scheduling graph) makes Lang's algorithm impractical for handling the workflows with an arbitrary dataflow DAG.

The root of these problems is that the resource-graph used in Lang's algorithm is constructed based on the process's control-flow information where the execution order of each resource node is pre-determined, which is not a general case in workflow-based computations. Therefore, in the case of workflow-based computations, Lang's algorithm is only feasible for the pipeline-shaped workflows where the job execution orders are pre-defined.

	Lang's	DAR	DTO
Resource-Request Graph (RRG)	a tree without integrating the scheduling information. The nodes along a path of the tree are executed one after another.	a general structured DAG integrated with scheduling information of the nodes. The resulting graph is called a scheduling graph, a tree on which the DAR algorithm acts to avoid deadlocks.	the same as the DAR algorithm
When to compute the maximum claim	statically computing the maximum claim associated with each node before the node is actually executed.	dynamically computing the maximum claim associated with each node during the computation. Thus, the maximum claim for a node is not fixed; it depends on how many nodes have not been completed.	the same as the DAR algorithm
How to compute the maximum claim	the localized maximum claim associated with a node is recursively computed by enumerating and traveling all its branches (rooted at the examined node) to select the largest maximum claim associated with its child node.	the maximum claim of each node is computed by exploiting dataflow knowledge to aggregate resource requirements of all the remaining nodes (i.e., those nodes that have not yet been completed).	the maximum claim of each node is computed by exploiting dataflow information to topologically order the remaining nodes in the DAG for the safety check.

Table A.1. Comparisons between Lang's Algorithm and our Algorithms: DAR and DTO

A.3 The Comparisons between Lang's Algorithm and our Algorithms: DAR and DTO

The key difference between our algorithms and Lang's is that our algorithms can handle the workflow graph (as well as its corresponding resource-request graph) with a more general structure. We first integrate the scheduling information with the dataflow DAG to define a scheduling graph, which is a tree to our algorithms and then compute the localized maximum claim associated with each node by leveraging the properties of this graph. For example, DAR computes the localized maximum claim by aggregating the requests of the nodes along a path, whereas in DTO all the remaining nodes are topologically ordered to compute this value. The detailed comparisons are shown in Table A.1.

Appendix B

Non-Uniform Distributions: Zipf-based Workloads

In addition to uniform distributions for job service times (JST) and file sizes, we consider the impact of other distributions on our algorithms. In particular, we consider the Zipf distribution, which has a large range, is highly skewed to the lower values in the range, and therefore has long tails in histograms of the value frequencies (Figure B.1).

In our experiments, the Zipf distribution generator is based on the algorithm presented by Devroye [22], but we limit the generated random numbers to the range [1, 500]. The Zipf distribution is characterized by a parameter $\alpha > 1$ to capture the amount of skew. We select $\alpha = 1.5$ for both the JST and file size. As with previous chapters, in these experiments, there are 100 instances in the workload and all of the workflow instances arrive at the same time.

B.1 Average of Simulated Makespans

Figure B.2 shows, especially for small storage budgets, how DTO has the lowest average simulated makespans when using Zipf distributions for both JST and file size. The advantage of DTO over the other algorithms is consistent with, but even more pronounced than with, average makespans and uniform distributions (Figure 4.14). In fact, there are a few data points with uniform distributions where DAR is faster than DTO (e.g.,Figure 4.14(b)), but DTO appears to dominate DAR for all data points with Zipf distributions.

As noted previously (e.g., Chapter 5.4.3), DTO tends to deal better with workflows where some jobs have considerably larger JSTs and file sizes than other jobs. By favoring already admitted

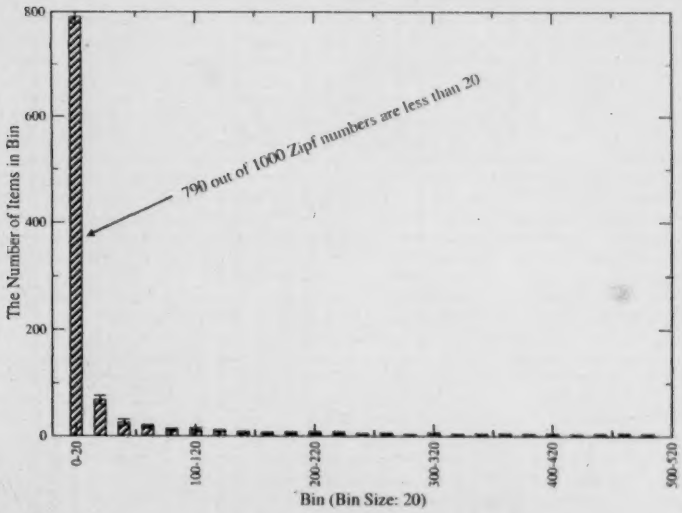


Figure B.1. Histogram of 1,000 Zipf Distributed Numbers ($\alpha = 1.5$, maximum=500)

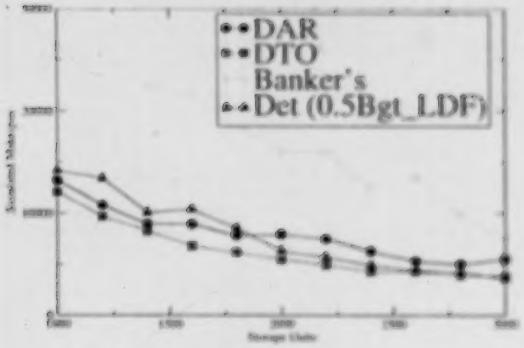
workflow instances over admitting new instances, DTO avoids the problem of inactive resource utilization.

B.2 Median of Simulated Makespans

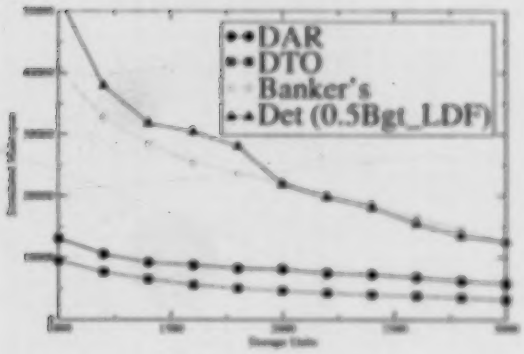
However, the skewed nature of the Zipf distribution also leads to high standard deviations over multiple simulated runs (Tables B.1, B.2, and B.3). Therefore, we also present the same 10 simulated runs of Figure B.2, but instead graph the representative data points using the median of the 10 runs (Figure B.3), instead of the average.

When considering medians, the advantage of DTO is less clear than when considering average makespans. However, DTO remains either comparable to the best algorithm in some cases (Figures B.3(a) and (c)), or the best algorithm in other cases (Figure B.3(b)).

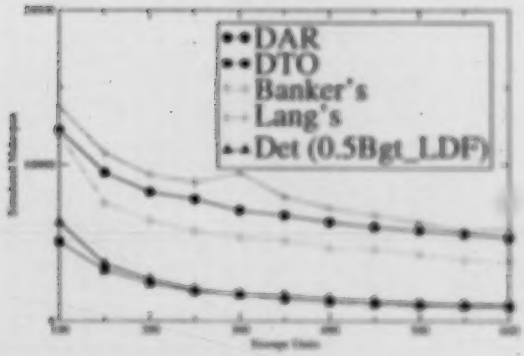
A thorough examination of Zipf and other non-uniform distributions is a topic for future work.



(a) Fork&Join (3×8)

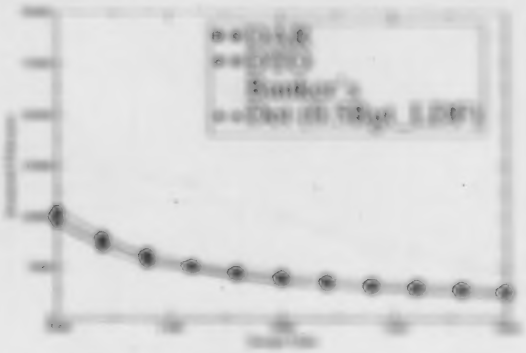


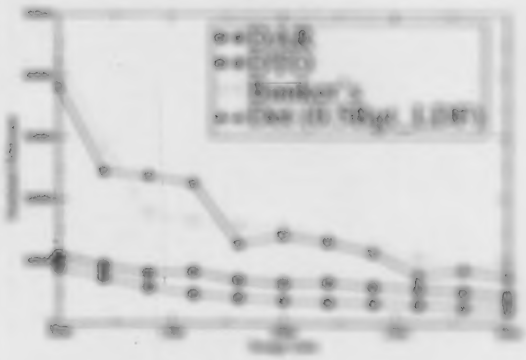
(b) Lattice (4×6)



(c) Pipeline (5-stage)

Figure B.2. Average Makespans: Zipf Distributions ($\alpha = 1.5$) for Job Service Time and File Size (10 Runs, Average)





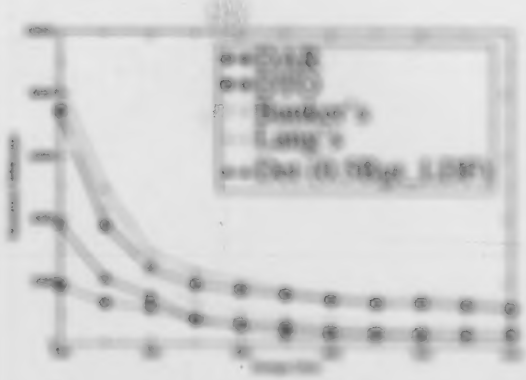


Figure 6.1. Median Makespans. (\(p\) Distributions (\(\cdot\) =) Hot-Job Service Time and File Size (10 Runs, Median)

Storage	DAR		DTO		Banker's		Detection	
	Mean (Dev.)	Median	Mean (Dev)	Median	Mean (Dev.)	Median	Mean (Dev.)	Median
1000	13195.4 (56.37%)	10572	12069.2 (65.1301%)	9444	26376.5 (73.9852%)	21020	14156.7 (103.348%)	9423
1200	10764.3 (58.28%)	8063	9670.6 (66.6141%)	7234	25677.3 (79.6774%)	14309	13395.2 (109.468%)	6947
1400	8951.2 (69.3062%)	5801	8292.8 (65.8857%)	6462	22518.7 (86.726%)	13957	10060.4 (80.5991%)	5381
1600	8915.7 (88.2864%)	5006	6800.8 (63.1433%)	5290	21600.2 (91.2554%)	13479	10345 (93.0438%)	4841
1800	7827.6 (98.6343%)	4258	6188.8 (69.0582%)	4721	18792.2 (92.4956%)	10288	8524.8 (87.4768%)	4158
2000	7949.3 (109.662%)	3746	5490.5 (63.6481%)	4309	16004.3 (90.32%)	10070	6315.3 (81.6924%)	3610
2200	7489.5 (125.861%)	3315	4969.6 (69.2454%)	3855	15985.8 (106.946%)	8640	5723.7 (89.4397%)	3309
2400	6285.8 (118.545%)	3036	4304.8 (66.6276%)	3464	12341.7 (97.18%)	6914	4922 (77.2362%)	2979
2600	5307.1 (104.244%)	2843	4298.6 (65.6382%)	3338	13460.1 (113.641%)	6678	4373.3 (79.2978%)	2756
2800	5021.5 (101.612%)	2627	3928.9 (63.9745%)	3059	9828 (112.431%)	5333	4219.8 (81.2795%)	2578
3000	5531.2 (118.008%)	2485	3814.3 (67.0394%)	2756	7486.1 (100.293%)	4793	3586.1 (69.0216%)	2361

Table B.1. Standard Deviations for Figure B.2(a)

Storage	DAR		DTO		Banker's		Detection	
	Mean (Dev.)	Median	Mean (Dev)	Median	Mean (Dev.)	Median	Mean (Dev.)	Median
1000	13094.6 (66.537%)	10886	9476.8 (47.225%)	8829	40676 (52.1543%)	42206	52413 (91.4224%)	38273
1200	10514.5 (46.3043%)	9293	7623.4 (47.9312%)	7353	32836.1 (54.5143%)	28210	37916 (71.9872%)	24374
1400	9169.6 (46.551%)	8064	6508 (51.0089%)	5823	28483.2 (63.0588%)	17837	31808.7 (70.4901%)	23652
1600	8729.4 (41.57%)	8263	5570.9 (48.1493%)	4788	25448.1 (64.969%)	16401	30366.6 (81.0315%)	22496
1800	8200.6 (46.7796%)	6888	5043.3 (48.2259%)	4180	23684 (63.4573%)	15681	28061.6 (99.6826%)	12684
2000	8064.6 (49.0271%)	6394	4686.6 (47.5396%)	3685	22069.2 (63.4802%)	15037	21897.9 (88.183%)	13974
2200	7408 (47.2951%)	6531	4260.3 (46.7345%)	3425	20252.5 (71.042%)	13559	19716.5 (97.2323%)	12927
2400	7209.7 (51.6111%)	5862	3970.3 (49.5066%)	3285	18614.7 (73.2416%)	11425	18137.8 (106.146%)	11244
2600	6750.1 (49.7191%)	5528	3769.9 (48.2363%)	3070	16355.3 (77.9709%)	10422	15457.1 (108.007%)	7659
2800	6170.9 (50.8373%)	5066	3477.4 (51.76%)	2601	14447.3 (79.1261%)	8723	13415.4 (101.329%)	8405
3000	5794.4 (53.7406%)	4441	3161.8 (61.0984%)	2392	12257.7 (80.2956%)	9275	12562.9 (104.658%)	7400

Table B.2. Standard Deviations for Figure B.2(b)

Storage	DAR		DTO		Banker's		Lang's		Detection	
	Mean (Dev.)	Median	Mean (Dev)	Median	Mean (Dev.)	Median	Mean (Dev.)	Median	Mean(Dev.)	Median
100	12254.1 (122.855%)	7444	5042.1 (166.096%)	1862	11955.4 (104.726%)	13777.8	8236 (108.78%)	7245	6297.5 (119.417%)	3806
150	9493.7 (161.016%)	3760	3257.4 (165.805%)	1330	7554.2 (131.496%)	10740.2	4938 (141.639%)	4254	3632.4 (136.123%)	2083
200	8246.3 (186.196%)	2423	2410 (157.267%)	1163	6405.9 (160.467%)	9353.6	2912 (165.217%)	2323	2656.3 (138.619%)	1520
250	7776.3 (198.812%)	1901	1893.9 (158.544%)	825	5715.4 (174.249%)	8867	2306 (176.356%)	1310	1980.2 (144.388%)	747
300	7062.8 (216.406%)	1731	1704.4 (157.16%)	582	5307.8 (187.139%)	9504.2	1978 (197.512%)	947	1672.5 (140.071%)	633
350	6744.2 (224.901%)	1605	1426.4 (158.597%)	372	5147.5 (197.95%)	7934.4	1708 (192.421%)	765	1607.6 (130.536%)	611
400	6312.8 (234.993%)	1434	1230.2 (162.196%)	363	4616.5 (209.955%)	7212.3	1525 (205.911%)	765	1406.3 (134.369%)	400
450	5982.8 (245.231%)	1319	1082.7 (165.407%)	335	4539.8 (227.287%)	6747.5	1435 (216.54%)	508	1254.7 (132.573%)	365
500	5781.2 (243.025%)	1332	1026.5 (160.184%)	329	4273.2 (234.423%)	6270.3	1463 (222.969%)	493	1097.6 (139.134%)	348
550	5552.5 (249.067%)	1251	967.5 (158.377%)	326	3907.8 (242.843%)	5694	1399 (242.249%)	489	1028.8 (132.879%)	326
600	5321.1 (251.989%)	1168	937.8 (164.16%)	323	3799.8 (255.926%)	5925.7	1194 (254.843%)	484	1045.5 (146.58%)	323

Table B.3. Standard Deviations for Figure B.2(c)